NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A DISTRIBUTED TIME BASED SIMULATION OF UNDERWATER ACOUSTIC NETWORKING USING JAVA

by

Brian S. Long

September 2006

Thesis Advisor:	Geoffrey Xie
Second Reader:	John Gibson

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Implementation of a Distributed Time Based Simulation of Underwater Acoustic Networking Using Java			**5. FUNDING NUMBERS**
6. AUTHOR(S) Long, Brian S.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved fro public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Underwater Acoustic Networks (UAN) have two immutable obstacles to overcome; the hostile environment in which it must operate; and the combination of the propagation speed of sound in water, and the latency in communication that this produces, and the dynamic nature of the water column with respect to its attenuation of the sound signal. These combined issues make it very costly and time consuming to setup a UAN just to test new protocols that may or may not be able to mitigate the limitations of this environment. There exists, then, a need for an ability to test a new protocol without the overhead of creating a physical UAN. The goal of this thesis is to provide a more hospitable, adaptable, flexible, and easily useable tool with which to test new protocols for UANs, as well as providing the ability for the Physics field to test new physical layer encodings. This simulation environment will provide the glue, or bridge, between the two disciplines by working as a common tool for both.

14. SUBJECT TERMS Underwater Acoustic Networks, High Latency Protocols, Delay Tolerant Networks, Time Based Simulation			**15. NUMBER OF PAGES** 115
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

IMPLEMENTATION OF A DISTRIBUTED TIME BASED SIMULATION OF UNDERWATER ACOUSTIC NETWORKING USING JAVA

Brian S. Long
Lieutenant, United States Navy
B.S., University of North Florida, 1999

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: Brian S. Long

Approved by: Geoffrey Xie
Thesis Advisor

John Gibson
Second Reader/Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Underwater Acoustic Networks (UAN) have two immutable obstacles to overcome; and the hostile environment in which it must operate; the combination of the propagation speed of sound in water, and the latency in communication that this produces, and the dynamic nature of the water column with respect to its attenuation of the sound signal. These combined issues make it very costly and time consuming to setup a UAN just to test new protocols that may or may not be able to mitigate the limitations of this environment. There exists, then, a need for an ability to test a new protocol without the overhead of creating a physical UAN. The goal of this thesis is to provide a more hospitable, adaptable, flexible, and easily useable environment in which to test new protocols for UANs, as well as providing the ability for the Physics field to test new physical layer encoding. This simulation environment will provide the glue, or bridge, between the two disciplines by working as a common tool for both.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost I thank Jesus Christ for my life, love and family from which all I drew strength during my time at NPS.

I also thank my wife Audrey for all her support. She is the one who listened intently as I complained about how many re-writes I had to do and always had the coffee ready in the morning after a late night of writing. I love you Audrey.

Professor Xie, you were often the reason for my wife having to listen to me complain but that was my fault, not yours. You stuck the bar high and made me cross it each time. I would be your student again. Thank you for everything.

Professor Gibson, you gave me your home and cellular numbers. How can I properly thank you for that? Our hours together were often long and tedious. You found all my logic and coding flaws and asked the stupid question when I forgot to ask myself. Thank you for your time and help.

Steven Brand, you are a friend among friends.

Bill Seegar, the office is not the same without you.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

Growth in remote maritime sensing and the employment of autonomous underwater vehicles has spurred the development and implementation of wireless underwater networks. Unlike traditional wireless networks, underwater networks use acoustic signals to carry data rather than electromagnetic (radio) signals. Many papers have been written regarding the use of acoustic signals to support underwater networks. Among these, Kilfoyle's provides a good synopsis of the issues pertinent to underwater acoustic network (UAN) deployment [1]. Subsequent papers by various authors tend to focus on the development of more capable signal processing techniques to improve signal quality and data rate, and thus improvements in modem technology, or report on various experimental UAN employments. Prominent research organizations in the United States involved in this research include Massachusetts Institute of Technology, Woods Hole Oceanographic Institute, Northeastern University, Florida Atlantic University, University of Washington, and Space and Naval Warfare Systems Center (San Diego). The Naval Postgraduate School activity includes underwater vehicle command and control research and access control protocol analysis by the Mechanical Engineering and Computer Science Departments, respectively. Various commercial entities are active in the development of acoustic modems necessary to implement the underwater communications channels.

UANs have two immutable obstacles to overcome. The first is the hostile environment in which it must operate. This includes not only the purely mechanical impediments of electronic equipment operating in water but also our ability as humans to work in this environment in order to deploy them. The second is a combination of the propagation speed of sound in water, and the latency in communication that this produces, and the dynamic nature of the water column with respect to its attenuation of the sound signal. These combined issues make it very costly and time consuming to setup a UAN to test new protocols that may or may not be able to mitigate the limitations

of this environment. There exists, then, a need for an ability to test a protocol without the overhead of creating a physical UAN, which would significantly hamper the progress of the testing.

The goal of this thesis is to provide a more hospitable, adaptable, flexible, and easily useable environment in which to test new protocols for UANs, which is the primary focus of the computer science field, as well as providing the ability for the Physics field to test new physical layer encoding. This simulation environment will provide the glue, or bridge, between the two disciplines by working as a common tool for both.

This thesis documents a simulation system that provides message distribution between simulated UAN entities. The system provides significant flexibility for modeling various protocols representing different layers of the UAN functionality. A key aspect of the model is its focus on reuse, where different protocol implementations or modeling techniques can be instantiated by the modeler in order to demonstrate or evaluate the performance of the protocols or techniques of interest without significant modification to the underlying message exchange mechanism..

In general, this thesis will provide an interface for researchers to test their layer protocols or models, such as link access protocols or signal propagation models by providing the simulation of a water environment and a definable UAN topology.

B. DESCRIPTION

1. Conceptual Layout

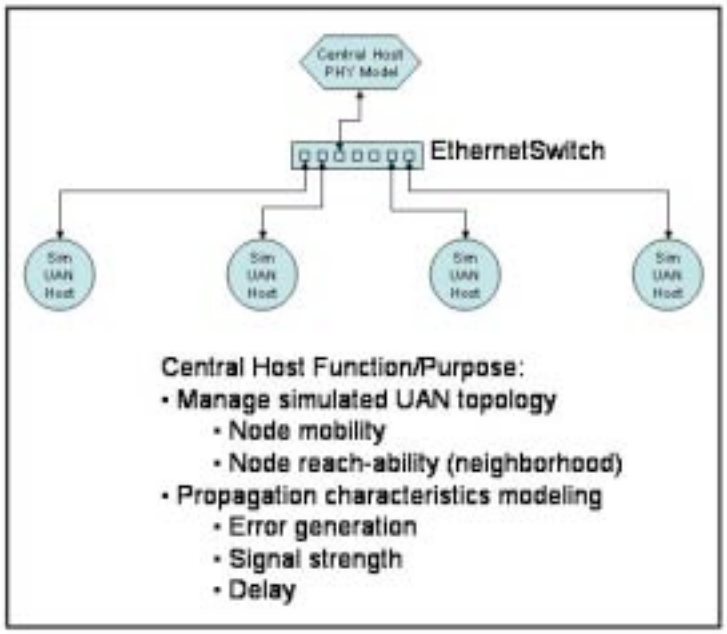

Figure 1. Conceptual Model

Figure 1 depicts a possible layout of the distributed model. Traffic is forwarded by the source node, via a TCP connection to the Central Host (CH), which models the properties of the physical medium and forwards the message to all appropriate simulated hosts. It is the CH that will also simulate the properties of the medium to be modeled. The utility of the CH is not related to the type of simulation but rather it has to do with omniscience. That is, it provides a god's-eye view of the topology, something the simulated UAN nodes can only assess by way of received traffic or protocol implementations. The Central Host maintains the modeled location of all simulated hosts in the physical topology representation.

2. Event Modeling

Some discussion could be had here as to the applicability of discrete-event or time-based simulation within the UAN. Although this is of concern in modeling any environment, it is not relevant to the outcome of this thesis. The environment itself

would not care on what metrics events are to occur and nor should not the simulation. As a physical layer message distribution simulation, this thesis is only concerned with passing data to and from the medium and upper layers. Any distribution of data beyond that falls into the functionality of higher levels of protocol and is left to future work.

3. Central Host Topology

As noted above, the forwarding of messages between a source and all recipients is by way of the CH. The message is then forwarded, by the CH, to all hosts that would be within range of the source. Since the physical medium forwarding properties will be unique for each UAN host pair, a separate session between the CH and each simulated host is appropriate. Further, since it is not desirable for messages to be dropped by the forwarding mechanism in the hosting environment, TCP connections are most appropriate and are used.

While the use of paired TCP messages (source-to-CH: CH-to-neighbors) adds a degree of artificiality to the overall model, its benefits outweigh that cost. Without a CH maintaining the god's-eye view of the modeled topology each node would need to maintain a local copy. While this would allow distribution of the physical layer calculations it would require careful coordination of topology status between all hosts creating opportunities for inconsistencies in the distributed network topology representation, especially in the case where mobile hosts are included. The CH method allows for ease of implementation, as sessions only need to be established between individual hosts and the CH rather than between all simulated hosts. Thus, adding or removing hosts will be less difficult than without a CH.

4. Distributed Host Topology

One method of overcoming the need for a god's-eye view, and thus the dominant need for a CH, would be for each node to periodically broadcast to all other nodes its current position. The timing of such broadcasts could be tied to the mobility characteristic of each node, thus stationary nodes would send a position update once (or very infrequently) while very slow mobile nodes would send a position update possibly every 500 milliseconds and fast moving nodes might send an update possibly every 100 milliseconds or less. If this method were to be used to eliminate the need for a CH then every simulated UAN host must maintain a communication session with every other host.

UDP may be used for this session so long as the update interval is sufficiently small to ensure that the loss of an update will not result in significant skews with regard to propagation delay calculations.

5. Implemented Topology

Careful selection of the physical medium model, either via a CH or distributed to each simulated host, has significant impact on the implementation of the model. If the artificiality of a CH for topology management is acceptable then the implementation of mobility and dynamic host availability is significantly simplified. However, a hybrid turns out to be much simpler, specifically in the calculation of collision detection that is needed to properly emulate the water acoustic medium and will be discussed in further detail later. This thesis implements a Central Host model with some functional calculations being performed in each UAN node. This distributes parts of the workload whose calculation results are unique to each node and significantly unburdens the Central Host.

C. HOSTING ENVIRONMENT

It was a goal of this thesis to create a simulation environment that would allow for more than one simulated node to run on the same physical host computer. An entire topology can be simulated on one computer with the power and resources of that computer being the only limits to the number of simultaneous simulated nodes. It is assumed, and recommended, that in larger topologies that the CH is run on a dedicated computer. In larger topologies, the CH will need sufficient resources to maintain TCP connections to all simulated nodes. The host systems were Pentium 4 based PCs with at least 500 MB of RAM.

To ensure guaranteed delivery within the hosting environment, a switched Gigabit Ethernet is used with the TCP connections between CH and host nodes. The propagation speed difference between acoustic signals and Gigabit Ethernet links provides the slack time for calculating message arrival times and collision events. This guaranteed delivery is essential and is the underlying assumption for the algorithms and calculations concerned with collision detection simulation.

D. THESIS STRUCTURE

The remainder of the thesis is organized as follows. Chapter II Background, provides the background of Underwater Acoustic Networking in general, this simulation environment, and the SAAM project from which it drew inspiration. Chapter III Simulation Design, details the design considerations for this simiulation. Chapter IV Simulation Implementation, details the specific implementation choices and explains the benefit or reason for each. Chapter V Conclusions and Future Work, is self-explanatory.

II. BACKGROUND

A. CHAPTER OVERVIEW

It is important that the reader understand the validity of this work. In order to ensure this, the history of UAN must be laid out. This will set the framework for discussions to follow and will include a definition of the problem, some participants in the arena of solving this problem and a selection of prior and current works in two different fields and how they apply to this thesis. The next sections will discuss the history, participants, work and direction, in that order and lay the necessary groundwork for the reader that is unfamiliar with the subject and explain the complexity and need for this work.

B. HISTORY OF UNDERWATER ACOUSTIC NETWORKING (UAN)

When was the concept of UAN first conceived? The only correct answer to that question is to acknowledge when the first person considered hearing anything under water. It may never be known if the first thought of underwater acoustic networking was from that one child intently listening to the thuds of his home while holding is breath underwater in the bath tub, or from the acutely aware teenager swimming in the river and hearing the levity of everyone around him. But it can be no later than the invention of the submarine.

The first submarine was a mere toy compared to the machines of today. Although William Bourne first drew plans for one in 1578, it was Comelius van Drebbel who took his rowboat and wrapped it in waterproofed leather and remained submerged in the Thames River for three hours [2]. But Drebbel was most likely so excited, that all he could hear was his own heart pounding within his ears.

The logical beginning then, would have to be at the time of the first military use submarine. This honor would go to David Bushnell in 1776. It was called the “Turtle” and was used during the American Revolution against British warships. From this point, submarine purpose and the need for underwater communication and surveillance continued to grow. It was the underwater telephone and then sonar that was the first venture into injecting man made acoustics into the water for a designed purpose. This

purpose climaxed with the start of the Cold War and the need to keep track of enemy submarines at long range from American coasts.

The Cold War, with all its submarine activity, sparked the first large-scale implementation of a UAN. The Sound Surveillance System (SOSUS) provided a detection capability of submarines using their faint acoustic signals [3]. With the end of the Cold War, priorities have changed and technology has advanced. Today threats and interest change too quickly to justify massive and stationary networks. Today's military and commercial advantages lie in quickly deployable, highly adaptable and dynamically changeable networks for vehicles, sensors and monitoring devices.

The research challenges described in the paper by Akyildiz, Pompili and Melodia, on Underwater acoustic sensor networks [4], offers a good overview of the UAN need and defines several uses in the following areas; oceanographic data collection, water pollution monitoring, offshore exploration, disaster prevention, navigation assistance and tactical surveillance.

C. WHO'S WHO IN THE UAN ZOO

It is not just militaries that have an interest in UAN, as do many international companies. Both have provided numerous grants to universities to conduct research for them, albeit more monies do come from a military interest as companies often worry about trade secrets and tend to conduct their own research. Nonetheless, these two large groups have asked for a lot of research to be conducted. This research includes sound propagation modeling, bandwidth utilization analysis, physical layer encoding, protocol analysis and many others in order to overcome the restrictions that are inherent to UANs.

Maybe the largest worldwide company conducting research in this area and providing solutions is the Sercel Company. Although a worldwide leader in seismic acquisition that continues to design, manufacture and provide for a full line of integrated equipment for, specific to this thesis, ocean-bottom cabling and marine environments, they acquired instrumentation in 2004 necessary for creation of an Underwater Acoustics Division. This division specializes in underwater acoustics and Marine Instrumentation for Offshore, Research and the Defense Industry. They offer a Multimodulation Acoustic Telemetry System designed to transmit over vertical and horizontal channels for

applications including bottom-fixed stations, AUV's, ships, bouys, and acoustic networks. Systems like this will be very useful once made easily deployable and not cost prohibitive, not to mention a little less proprietary.

There are also several universities that conduct extensive research in acoustic propagation models and underwater networking. Along with the Naval Postgraduate school there is The University of Texas at Austin, MIT and Georgia Institute of Technology, just to name few. All of these universities have conducted, or are currently conducting, research for various military interests from SPAWAR, NAVSEA, ONI and ONR. There were also at least two experiments conducted mainly by military interests and were the Fleet Battle Experiment-India (FBE-I) [5, 6] and the Autonomous Undersea Systems Network (AUSNET) [7].

The ASW experimentation for FBE-I, using Seaweb, focused on undersea acoustic network connectivity between the acoustic and radio frequency regimes, while incorporating existing ASW data formats into a theater tactical capability. FBE-I was the ninth in a series of CNO sponsored experiments coordinated by the Navy Warfare Development Command (NWDC). The NWDC Mission is to "Operationally examine innovative concepts and emerging technologies to identify advanced war-fighting capabilities for further development and rapid transition to the fleet."

The AUSNET is a project that addresses the need for ad-hoc self-forming networks that can operate in low-bandwidth undersea environment. The AUSNET is a STTR Phase 2 program addressing network protocols for undersea communications that is jointly sponsored by the National Science Foundation and the Office of Naval Research. Applications for underwater networks are limitless. The ability to form an acoustic network from different platforms, ranging from stationary sensors to unmanned autonomous vehicles, to submarines and surface ships, would allow a richly interactive environment for data collection, surveillance and data distribution.

Many interests have been shown and can be grouped into three major areas; modem design, acoustic propagation equations and networking and protocol design, but can also be broadly placed into two general categories; that of the acoustic engineers and that of the computer scientist. The acoustic engineers are concentrating on the proper modeling of sound propagation in order to develop modems that can operate in all underwater

conditions, while the computer scientists are concentrating on simulations and protocol efficiency in order to develop a network that can operate in all underwater conditions.

The problem then becomes that one does not directly support the other. The problem of underwater acoustic networking is currently being attacked from two distinct and often disjoint disciplines with no foreseeable meeting in the middle. It is correct to work on a problem from more than one angle, but knowledge of these angles may help each other and will only yield the best picture for solution when both are used together.

D. WORKING THE PROBLEM FROM BOTH ENDS

To work a problem from both ends, the ends must be defined. On one end there are the acoustic engineers with propagation equations that can take several hours to calculate. On the other end there are computer scientists with network simulations and protocol analysis running in discrete-event, compressed time. The problem is that one does not compliment the other. A calculation that takes several hours cannot be used in a real-time simulation and a simulation that can not be run in real-time, does not support protocol analysis in a truly dynamic environment. Currently, shortcuts are being taken, by both sides, which make the validity of their results questionable.

For underwater networking to work, the impediments to the networking aspect must be solved. The research done by Xie, Gibson and Yang [8] suggest that both dedicated access and pure contention access offer improved network performance over collision avoidance techniques currently in use. However, this research cannot be completely validated without a simulation capable of modeling the time dependent and dynamic nature of UAN traffic. Tools are needed that can offer this simulation environment and model acoustic propagation with the accuracy of the equations from the acoustic engineers and the efficiency of the protocols from the computer scientists. This accuracy in propagation is necessary to properly evaluate the needed protocols to make underwater networking a sustained reality. This sustained reality means reliable delivery and usable bandwidth for a reasonable cost. The tool does not currently exist that can bridge the complexity of the acoustic model offered by the acoustic engineer and the protocol abstraction of the network offered by the computer scientist. However, several validation attempts have been made at the edges.

Two examples on the acoustic side are the Navy Standard Parabolic Equation model and the Monterey-Miami Parabolic Equation (MMPE), while two examples on the computer science side are offered from Sozer and Coelho. The Navy equation was developed by the Naval Research Laboratory (NRL) and is implemented as the standard acoustic modeling for the Navy [9] while Kevin Smith and Fred Tapper [10, 11] developed the MMPE. Both of these models are very robust, very specific to an area and very computationally intense, which does not lend them to real time simulations. On the other end, there is the design and simulation of an underwater acoustic LAN by Sozer, et al. [12] and an analysis of medium access control scheme for UANs by Coelho [13, 14]. Both severely simplify the propagation loss calculations, the first using frequency and background noise level, and the second using only static range. These models are effective for use in a network simulation for course protocol testing but, without the true propagation model representation, fine-tuning and truly accurate protocol testing cannot be performed and are typical of other solutions being offered. This remains the current shortcomings of the tools in this area as the divide of models and tools offered by acoustic engineers and computer science simulations remains un-bridged. That is, for propagation models to not overload the simulation the equation must be unduly distilled and simplified, or for a simulation to run the full calculations required by a precise model, it will not be able to run in real time. This leads then to the debate over Event based or Time base simulations and the pros and cons for each. Event based simulations could operate under these restrictions where time based simulations could not.

Event based simulations are often very appropriate where performance of a protocol is desired, since the timing can be compressed. But no matter how useful they are in time utilization, discrete event simulations do not allow for input from external sources, only that of a pre-scripted simulation timelines. These external sources can be that of simulated or real sensors, vehicles or agents. These time constraints and real time interaction are not mandatory to have a working and complete model, but they are significant enhancements over existing tools and will aid greatly in future developments to solve the complex model of underwater acoustic communication. Also, it is desired to create hybrid environments of simulation and physical networks for extensive testing of physical modems and network protocols. Event based simulations can not accommodate

this requirement as events in real time are not always known and can not be planned for in a discrete manner. Therefore, a time-based simulation with external inputs is needed.

Modeling underwater acoustic network message distribution is a severely complex problem. The previously mentioned paper [8] gives two particular issues that bear heavily on the problem: (1) The variability of the channel impulse response, which requires evaluation of the channel for signal propagation characteristics for each message transmitted; and (2) the extreme propagation delays characteristic of UANs, as compared to wired or radio-based networks, which complicates the detection and handling of message collisions. Both of these issues are dynamic and thus affect the medium in real-time, thus requiring a time-based model for evaluation that is also capable of taking input from external sources for dynamic attributes or propagation models. It is these external sources that will allow for the correct characteristics of the dynamic medium to be simulated in real time.

Problem (1) above is complex and as suggested in [8], will likely result in the use of a server farm to obtain a timely result. To utilize this server farm, the capability of external input is required. A previous thesis by Diaz-Gonzalez [15] has addressed the issue of simplifying this calculation but to also retain a more realistic propagation model than those proposed by Sozer and Coelho. This simplified method is a compromise between calculation precision and the processing power required to calculate it. It is able to be used directly within a simulation with promising results and takes the process much closer to reality. This calculation, however, is still limited in its ability to allow for changes during testing and does not support the dynamic need of physical layer designers.

Although problem (1) above appears to be solved, at least in a limited but usable manner, it is not extensible to a mobile topology simulation or that of one that is part simulation and part actual physical network. That is, it will work fine for completely static topologies of stationary nodes in calm waters. This does not allow for mobile nodes or more than minor node variances in stationary node position, as shown in [15] to cause significant temporal differences. Also, using Sozer's or Coelho's methods would seem to solve problem (2) above but only succeeds in providing a gross estimate of the

solution. To obtain an accurate evaluation of a networking protocol or coding scheme, each must be tested within the effectiveness of the other. This can only be accomplish within a simulation tool that allows real time dynamic acoustic propagation modeling as well as network protocol testing.

A flexible tool, taking external inputs, would allow the testing of new physical devices as well as multiple protocols by integrating the two disjoint knowledge areas of signal processing modeling of underwater acoustics and time-based computer network simulation. It is this capability that this thesis hopes to capitalize upon and limit the need for expensive field experiments.

E. DIRECTION

As laid out above, neither the acoustic engineer nor the computer scientist can solve this problem of underwater networking single handedly. This system concept has been described in detail in [8] but with no current implementation. This thesis is the first attempt at implementing the previously discussed model. This implementation of a testing environment, a useful tool, or a simulation that can use the precise propagation equations currently available; that can run in real time to evaluation how a protocol can handle all situations and be able to take input from the real world, or that of physical networks that are involved in the testing, will allow researches from both sides to quickly integrate and evaluate theirs ideas together. This tool could be the glue between the two communities of physics, with its acoustic engineers and modem designers, and that of computer science, with its network and protocol analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESIGN

A. CHAPTER OVERVIEW

Unlike traditional wireless networks, underwater networks use acoustic signals to carry data rather than electromagnetic (radio) signals. Unfortunately, the underwater acoustic environment induces severe limitations on the usability of a UAN. Two of the key limiting factors are severe propagation delays, associated with the approximately 1500 meters per second propagation rate of the acoustic signal in seawater, and constrained bandwidth, due to extreme attenuation of frequencies above 50 Kilohertz over any appreciable distance, normally about 1000 meters [16]. Further, the Omni-directional nature of acoustic water channels leads to the same hidden terminal, exposed-station, and near-far problems associated with traditional (radio-based) wireless communications.

To address these latter issues, a wireless network typically implements a form of collision avoidance access control to coordinate the transmissions of multiple users. Such techniques employ the exchange of "handshake" messages, which grant the use of the medium to a single user while informing other users within range of the intended recipient of the pending transmission. One such example is the RTS-CTS used in the 802.11 protocol. But this "handshake" creates significant overhead, in a high latency medium, that dramatically reduces throughput of a UAN. This then requires new protocols to be created and tested.

This chapter will investigate the different design options to best solve the problem of simulating an acoustic water medium and how to simulate message handling and collision detection. These options should provide for as much distributed processing and functionality as possible while providing the user with a clean and diverse tool in which to run protocol simulations.

1. Key Components

There are two distinct functions to consider, Simulation Control and Physical Layer implementation. Simulation control consists of setting up TCP communication sockets between nodes and providing an interface between channel simulation code and

the physical layer model implementation. Physical Layer implementation consist of properly representing the acoustic behavior in a given water column between two points. This acoustic behavior, or Acoustic Propagation Model, also depends on the characteristics of the sound, and therefore, the physical device (or acoustic modem) characteristics used for underwater communications.

These two functions are not mutually exclusive. In order to provide the least amount of latency induced by the implementation, it will be necessary to tightly couple and streamline the two functions. This will most likely produce an implementation that can not be separated functionally but is a sequence of highly optimized, in-line code. This will provide a tool with minimal simulation overhead and be able to run on the most standard of current computers.

There are also two logical environments to consider to aid in understanding the intricacies of this work, the simulated environment and the hosting (supporting) environment. The simulated environment is just that, simulated, and consists of just the network topology created by the user and its relation to the water medium. The hosting environment is the actual computers and network on which the simulated environment is created and run. This can be as minimal or expansive as the user sees a need to run. Several points of implementation are addressed later, as a result of the difficulty of simulating a water medium acoustic environment.

2. Possible Design Models

The purpose of this work is not to implement any protocols but to create a simulation in which to test protocols for a high latency medium. This simulation must emulate the water medium in as many aspects as possible. The better the simulation the more reliable the results will be.

One goal of this work was to create a simulation that could run on the most common of computer networks. This would allow for continued use of previous investments and expedite simulation setup and usability. There are two distinct models that would accomplish this goal, the centralized model and the distributed model.

The centralized model is a viable option only if a high performance computer is available or the simulation is not too complex in network topology, measured by the

number of simulated nodes in the network. A single computer can host the Simulation Control and the simulated Nodes as long as the overhead of processor state switching does not inject an artificial latency. A centralized model would also have perfect knowledge needed for collision detection; however, it would require a rather large structure to hold the many neighborhood conditions that relate to collision detection described later in this chapter. This model is ideal with respect to user interface, ease of setup and running, topology view and time synchronization but may require an unusually powerful computer on which to run.

The distributed model is a more viable option to run across an existing network and investment, which satisfies one goal of this work. The structures of collision detection are much simpler when only related to one simulated node. Here the distributed model makes better use of existing resources by not requiring specifically high performance computers. This model would not be as easy to setup, as the simulation would need to be loaded for each simulated node. This requirement would basically move a function of the centralized model to the user of the simulation and require him to coordinate the loading of information to each simulated node. This would also apply each time any changes to the topology where made. This model is ideal only in the matter of using lower cost networks that may already exist but would require more complex implementation in terms of the actual simulation core implementation, and loss calculations without a central topology view being held to determine propagation losses.

3. Design Model Used

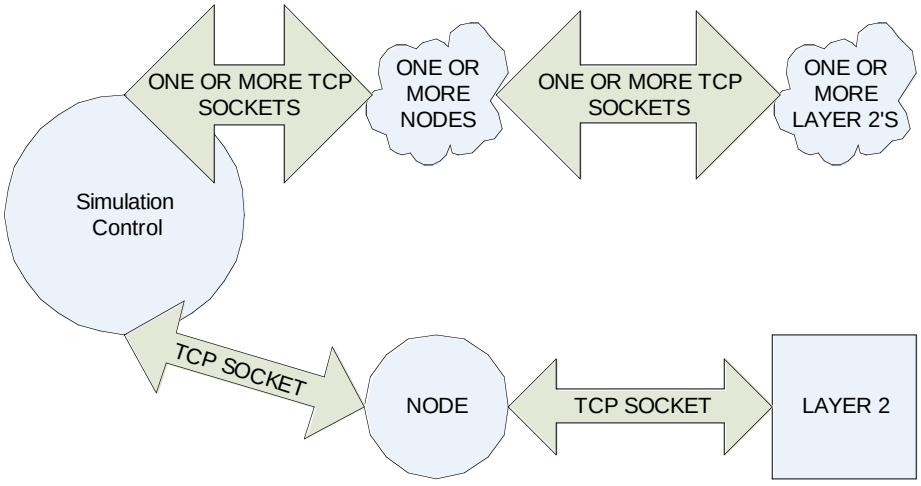

Figure 2. Hybrid UAN Model

Network topology view is much simpler to maintain as a gods-eye view and is incorporated in the centralized model's functionality resident in Simulation Control (SC). This gods-eye view also leads to incorporation of the propagation delay and loss calculation functionality into SC. Since it is much simpler to maintain topology information in a single location, it is also simpler to maintain neighborhood information in the SC and, thus, the propagation delay and losses from any node to any other node. The SC also remains the central point for Layer 2 Application connection requests. It is assumed that the SC location will always be known and that this simplifies this sequence of events.

To ensure continuity between different programming languages that might be used as a layer two process, an importable object is provided that, once instantiated, will handle the simulation connections for that user as well as translate data from format of the message forwarding layer, analogous to the acoustic physical layer, to the Layer 2 process format and vice versa. This interface is defined in the use of two method calls to the interface object, *send* and *receive*. The *send* method allows the layer two process to pass data to the simulation while the *receive* method allows the process to retrieve data from the simulation. This interface object handles all data transformation and assumes that any process that instantiates it, can operate with byte arrays. The data format, to and from a layer two process is standardized as byte arrays with the format conversion to and from the simulation being handled completely by the interface object internally. The data format change, from the layer two process to the simulation and vice versa, is handled within the interface object and will receive from and pass back to the layer two process, a byte array. It is reasonable to assume that all programming languages that might use this interface can operate with byte arrays.

The Node portion of the simulation can run on the same physical host as SC, or a separate computer, and performs all collision detection since all aspects of a message collision is relative only to the receiving node. The simulated node is the only point of connection and communication to and from the layer two process, while also maintaining a separate communication connection to the SC for data flow within the simulation. This unburdens the SC and distributes processing power requirements over as many physical machines as are available to run the simulation.

4. Definition of "channel"

For further discussions, the term "channel," will refer to a set of unique acoustic conditions in which communication between two Nodes occurs. The combined effect of the unique properties of the water column are most directly applicable to this research in terms of link propagation delay; acoustic signal loss, and thus effective range; and link bandwidth. Hereafter, we consider these three values as they affect this simulation environment the most and may be the data returned from an external calculation.

B. DEFINING THE OBSTACLES

Simulation of a wireless environment within a wired hosting environment highlights several issues with respect to how to actually simulate propagation and transmission delays, and the proper representation of collisions within the physical medium (collision detection). While the computations of the acoustic propagation, which will provide channel loss values, are complex, these calculations have already been developed and may be computed internally or externally, as suits the goal or purposes of the particular simulation user. Physical Medium Collision Detection (PMCD), however, is handled within the simulation.

Collisions, in the physical medium, result when a receiver receives more than one signal, at any given instance in time, without the ability to distinguish between each of the different signals. And although a different mechanism, the same networking condition results when the receiver is blanked, due to its own transmission status, during a time that a signal was present in the medium to be received. In the actual environment, both of these conditions would simply result in either no signal or a garbled or unrecognizable signal that may or may not get passed up to the upper layer depending on implementations of error correction and encoding within the physical layer being simulated. It is these conditions that must be represented correctly to properly simulate the water medium.

1. Propagation Delay

Propagation delay in water creates an issue with current protocol timing schemes. With the nominal speed of sound through water of 1500 meters per second, and the typical UAN range of 1000 meters, there is a one-way propagation delay of 667 milliseconds, as illustrated in Figure 3. That is, the time between when a bit of data

leaves a transmitter until that same bit of data reaches the receiver 1000 meters away, is 667 milliseconds. This is more than 4 times the average RTT of the Internet today.

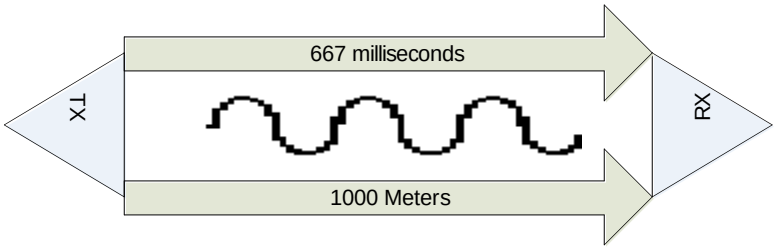

Figure 3. Propagation Delay at 1500 m/s

As discussed in earlier chapters, loss calculations can be intense, due to acoustic propagation model complexity, and not typically suitable for a time-base simulation. This then requires the use of less precise models of propagation, also discussed previously, to maintain this time-based simulation within the bounds of the minimum propagation delay as defined by the network topology and acoustic conditions where this network is located. In the simplest form, propagation delay is a function of sound propagation rate and distance. The loss calculation determines the possible reception range of a signal; given a particular minimum receive signal level or Signal-to-Interference/Noise-Ratio (SINR).

There are three possible design scenarios to accomplish these calculations. The first is that all calculations are completed prior to allowing any simulation traffic to flow. This would allow for the more complex calculations but would limit the acoustic channel conditions to a static state for the duration of a given simulation. The second is that less complex calculations are computed for each message to be sent within the simulation. This method allows for a limited dynamic set of acoustic channel conditions. This option would only be limited by the computing power of the physical hosting computer on which the Simulation Control is running. The third is for calls to a propagation modeler outside of this simulation. This would allow for the use of massive external processing power, possibly a server farm, to provide the answers to the complex propagation models within a time bounded by the requirements of this time-based simulation. The last method would also allow for referencing experiential data catalogued at various research agencies.

This work implements the first method, static calculations prior to simulation, but incorporates a completely modular design so that the second or third method can be implemented in the future by only changing one method within this simulation. This calculation provides the loss values and propagation delay to the SC. This is handled strictly within the SC where the neighborhood topology is formed based on the loss value returned; and a packet is not sent to the destination until the propagation delay has elapsed.

2. Transmit Time

Transmit time is tracked and used to calculate the beginning of the collision window. This tracking is only needed due to the difference in the speed of the hosting environment and that of the simulation environment, due to the high latency of the water medium being simulated. It is possible that multiple packets will be received, in the hosting environment, during the simulation time of even one message. This time is the starting point for calculations of propagation delay, transmission rate, transmission window and collision window. It is assumed that all hosting computers used in the running of a given simulation, will have system times synchronized in some matter. This allows for the transmit time of each packet to be set to the time that the simulation receives it from the layer two connection. This packet transmit time is then checked by the SC and modifies the simulated propagation time by any delay that might have already occurred due to communication lag from the Node to SC. This communication lag is assumed to be negligible due to the magnitudes of difference in speed between the hosting environment and the simulated environment but is implemented in this way to allow for use of slower hosting environments when necessary.

3. Transmission Rate

Transmission rate can be different for each channel defined and therefore must be accounted for and tracked for each packet flowing within the simulation. This is accomplished with the use of channel IDs and is set within each packet, by the simulated node that transmitted it as to which channel a given packet was transmitted on. If a receiving simulated node can receive on the channel that a packet was transmitted, then it will know the rate and calculate it accordingly. Transmission rate, with message length, defines the duration of the collision window.

4. Collision Detection

The Collision Detection Finite State Machine (FSM) represents the algorithm of collision detection used in the *CollsionDetectionThread* class, as illustrated in Figure 4. The idle state represent that no messages are currently being received. This can be that no messages have been received, as would be the case at the start of the simulation, or that the collision window has expired. The transition from idle to busy occurs when a message is received. The busy state represents a message currently being received that has not collided with any other message. The transition from busy back idle occurs when the collision window expires and no collision has occurred and the message is passed to the upper layer. The transition from busy to collision occurs when the collision window has not expired prior to another message being received, thus a collision has occurred. Any other messages received while in the collision state will cause a transition back to itself and remain in the collision state until the collision window has expired. The transition from collision to idle occurs when the collision window has expired and all collided messages have been dropped.

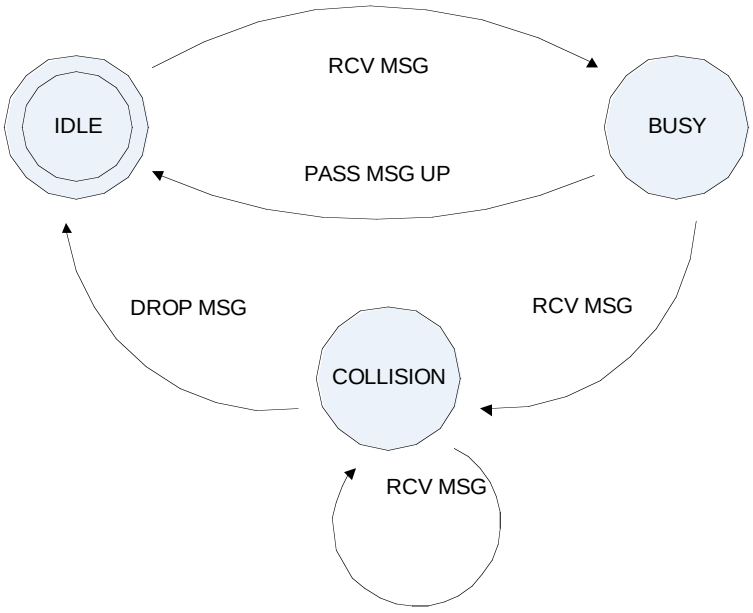

Figure 4. Collision Detection FSM

The core coding and implementation of the physical layer collision detection was a significant obstacle of this work. The proper representation of a physical collision is

imperative for a correct and useful simulation. To determine if a collision has occurred it must first be determined if any receive window overlaps another receive window or that of a transmit window for a specific channel, as illustrated in Figure 5.

An example of a four message situation might be as follows:

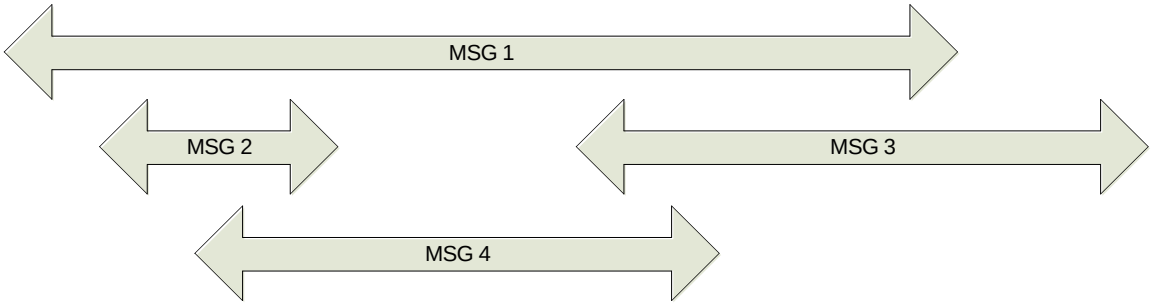

Figure 5. Collision Window Overlap Per Receiving Node

Message 1 collides with messages 2, 3 and 4; message 2 only collides with messages 1 and 4; message 3 also collides with messages 1 and 4 while message 4 collides with messages 1, 2 and 3. As can be seen from the figure, if any message reception begins after the start of another message but before the end of that message a collision occurs and both messages are subject to corruption, and thus rejection.

To calculate a collision window, the transmit time, transmission rate, size (length) of the signal and propagation delay must be known. Transmission rate and propagation delay are channel dependent. The collision window is a duration calculated from the length of the signal and the transmission rate of the channel. The transmit time with the propagation delay, defines the start of the collision window, relative to the receiving node. That is, if two nodes transmit the same signal, on the same channel, to the same destination node, at the same time; if one of the transmitting nodes is far enough away from the intended destination node, then the two signals will not collide and will be properly received. This is true due to the fact that propagation delay, in its simplest terms, is a function of the speed of sound through water and the distance to travel. So, as long as the propagation delay for one of the messages is greater than the sum of

propagation delay and transmission delay for the other message, then the two messages will not overlap and, thus, will not collide; as illustrated in Figure 6.

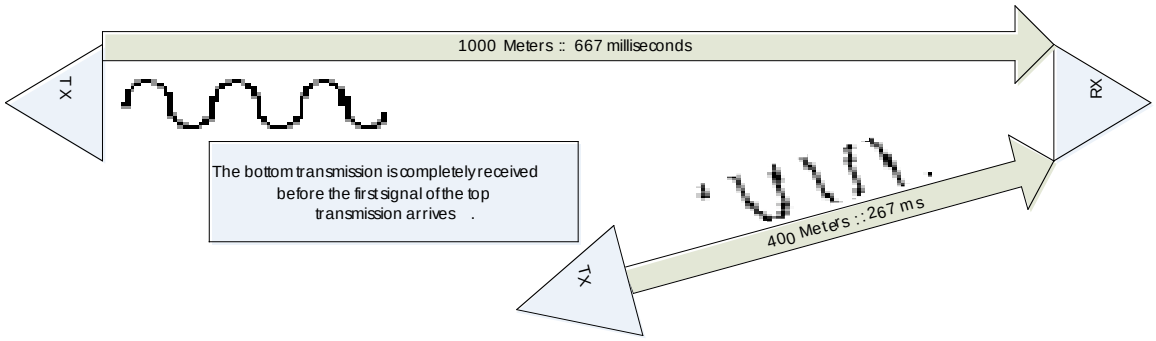

Figure 6. No Collision with the Same Receiver

C. STRUCTURE

For this simulation environment, functionality is divided between the Simulation Control (SC) and the simulated node. The SC is the focal point of control, simulated node communication and medium emulation, as well as conducting the UAN topology setup. The SC also accepts requests from applications to connect to the simulation acting as a layer two or higher protocol. A simulated node maintains communication with the SC and a layer two protocol application and performs collision detection with respect to messages sent and received. This division of labor, intended to be as distributed as possible, is further defined below.

1. Functionality of the Simulation Control

The SC handles the following functions: reads a network topology from a file; estimates propagation loss (internally or externally); determines each nodes neighborhood topology from the network topology and channel loss estimates; establishes and maintains communications with all simulated nodes; handles request for layer two connections; and simulates propagation delay.

a. Reading in the Network Topology

Once the user selects the file to read, the SC parses this XML file and builds the network topology for the simulation and stores this information in a network information object.

b. Setting up the Neighborhood Topology from the Network Topology

Once the network topology has been created, all node locations are known and all nodes are compared to all other nodes based on channel loss calculations. If it is possible for a node to hear the transmission of another node based on the estimated signal range as derived from the propagation loss estimate, then it is considered in the neighborhood for that node. This relation will not necessarily be symmetric as acoustic channel conditions are not bi-directional and may prevent communication in one direction between a pair of nodes. Further, depending on the complexity of the loss calculation employed, this value may be dynamic and result is periods of discontinuity between neighboring hosts based on such factors as node mobility or wave motion.

c. Establishing and Maintaining Communications with each Node within the Simulation

The SC will wait for connections to be made from as many nodes as are defined in the network topology. Separate communications are maintained for each simulated node. Once all connections are made, simulation setup continues with each node being assigned its identity within the simulation based on the network topology defined.

d. Handling Requests for Layer 2 Functionality and Connection to the Simulation

A Layer2 Application sends a request for connection to the SC. The SC then forwards this information to the next available simulated node.

e. Simulating Propagation Delay

Loss calculations are determined from the propagation model and may be different for each direction of communication between each pair of nodes within a neighborhood. The SC will not forward a packet to a node until the propagation delay elapses, to ensure that the packet arrives at the node at the time that the first bit of the message would have arrived, as if in the actual environment being simulated.

2. Functionality of Nodes

The simulated node maintains communication with the SC and a layer two process. The simulated node also performs collision detection. If a collision occurs, the message is not passed up to the layer two process.

a. Handling Communications with the Simulation Control

Communication with the SC is maintained with a TCP socket for guaranteed delivery and reliability.

b. Handling Communications with the Layer 2 Process

Each simulated node maintains communication with a layer two process using a TCP socket for guaranteed delivery and reliability. The layer two side of this communication is actually an interface object, provided by this work that encapsulates the TCP socket and provides a straightforward access to the socket's stream objects. The layer two process actually sends and receives its data via this interface object, that it imported and instantiated as part of its application.

c. Performing Physical Layer Collision Detection in the Simulation in Order to Properly Represent It to the Layer 2 Process

A Collision Event is simply defined as two overlapping receipts or a receipt during transmission on a simplex link. In either case, this model will not pass a message up to the layer two application. The key factors here are Transmit Time, Transmission Delay and Transmission Window.

(1) Transmit Time and how it affects CD. The transmit time defines the start of this sequence of calculations as it starts the clock for the transmission window, receive window and collision window.

(2) Transmission Delay and how it affects CD. Transmission Delay defines the duration needed by a receiver to receive a signal once the first bit arrives. It is a function of transmission rate of that channel and the received message size and defines the duration of a collision window.

(3) Transmission Windows and how it affects CD. The Transmission Window only affects CD when a channel is designated as a transmitting channel. If a channel is able to transmit and receive, the transmit window becomes a collision window for its duration.

D. COMMUNICATION

The two commonly known models for communication are in-band and out-of-band communication. The out-of-band communication would require a second socket with which to communicate with each node. The overhead, of a second socket, was not desired. The serialized nature of network simulation setup and then running of the

simulation was very compatible with in-band communication, which also minimized the use of resources within the hosting environment.

Simulation communication uses a serial adaptation of the in-band model. All communication originating from the SC (i.e. the setup) is accomplished first. Once the simulation is completely setup, this type of communication discontinues. The environment then runs purely as a simulation until termination. Communications from SC-to-Node, Node-to-SC, Node-to-Layer2 and Layer2-to-Node are all via TCP sockets. The TCP sockets provide a reliable delivery scheme to build the simulation upon.

E. USER INTERFACE

A GUI interface was considered but would have been utilized too little for the effort. Very little user input of data is required and is more effective to use a command line interface.

Inputs required from the user are via command line and are as follows: the IP address and port number of the SC upon launching a Node; and the IP address and port number of the SC upon launching any application that instantiates the *UAN.Helpers.Layer2Connect* object to connect to the simulation.

One GUI is used for the selection of an XML file that contains the network topology of the simulation, which is an input to the SC.

F. DIRECTION

This chapter has laid out the pros and cons of two different design models, the centralized and the distributed model. The strengths of both models were chosen to form a hybrid approach that will now be implemented and detailed in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION

A. CHAPTER OVERVIEW

The goals of this implementation were modularization, abstraction of message passing, distributed processes and functionality, non-blocking I/O and minimizing simulation overhead. Modularization of code provides for functionality that can be easily upgraded and enhanced when needed. Abstraction of the passing of information within the simulation, as objects, ensures that the underlying handling structure is not dependent on the type of messages to be handled. To ensure this tool can effectively run on the computers available in the most common network environment, it is necessary to distribute the processing power required and as a result the functionality, to lessen the maximum requirements of any computer to run this tool. This also helps to minimize simulation overhead. Simulation overhead is also considered any time a loop is required. To ensure that processes and threads do not induce undesired overhead, they are forced to yield when performing a busy-wait condition, such as a while-true loop.

One significant issue is I/O. In order to guarantee message delivery, even using TCP connections, it is necessary to implement a non-blocking I/O scheme. Java provides for this in the *nio.channel* package since JDK 1.4.2, but to ensure as little simulation overhead as possible, this ability was implemented simply, with minimal code, using separate threads for each stream of a given socket.

Modulation and abstraction was also enhanced by using Java as the programming language. The main reason for using Java, however, was to provide maximum portability and cross platform compatibility for this simulation tool so that its use would be as seamless as possible. The functionality of the Simulation Control (SC) is handled within the *SimControl* class while the Node functionality is handled within the *NodeControl* class. This chapter describes the classed developed under this thesis effort to implement this simulation.

This work was coded in the NetBeans 5.0 IDE and is broken into four packages, *UAN.Channels*, *UAN.Sim*, *UAN.Helpers* and *UAN.Node.*

B. CHANNELS PACKAGE

The *UAN.Channels* package contains the channels which are defined for a given simulation. The package contains four java classes, *Channel.java*, *ChannelZero.java*, *ChannelOne.java*, and *ChannelTwo.java*.

1. Channel.java

This class is a superclass for all channels and defines the minimum attributes of a channel needed to support the simulation. This class is constructed with; a name as a string, the transmit rate as an integer, the frequency as an integer, and the unique channel ID as an integer. This class also contains four public methods to return these values when needed.

The subclasses of channel were only used to test the simulation. During simulation testing, things were kept simple and easy to calculate by using the same transmission rate for all channels. It is possible that all channels are identical with the only requirement being that each channel in the simulation has a unique channel ID. This allows for an over simplification of a channel to support simple simulation runs by allowing unique channels within the simulation that may be identical in the attributes that affect the simulation. That is, all of the transmission rates might be the same, but only the frequency or underlying encoding is different. If transmission rates did not change, then the collision window duration did not change and collision detection is not affected. However, the frequency and encoding would affect the loss calculation provided from the acoustic propagation model; but those calculations would have already been determined.

Constructor	`public Channel(String name, int transmitRate, int frequency, int ID)`
Methods	`public int getTransmitRate()`
	`public int getFrequency()`
	`public String getName()`
	`public int getID()`

Table 1. Channel

a. The getTransmitRate Method

This method returns the transmit rate of this channel as an integer.

b. The getFrequency Method

This method returns the frequency of this channel as an integer.

c. The getName Method

This method returns the user defined name of this channel as a string.

d. The getID Method

This method returns the unique ID of this channel as an integer.

2. ChannelZero.java

This class extends the *Channel* class and is constructed by the user with only a name. This channel has a 1 Kbps transmission rate, a frequency of 4 Hz and a unique ID of 0.

`Constructor`	`public ChannelZero(String name)`
`super call`	`super(name, 1*Kbps, 4, 0)`

Table 2. ChannelZero

3. ChannelOne.java and ChannelTwo.java

These two channels are identical to *ChannelZero.java* except for ID which is unique. These channels were built minimally to allow the simulation to run. It is expected that once this tool is utilized, that this package of channels will be enhanced.

C. SIM PACKAGE

The *UAN.Sim* package contains seven classes; *SimCommsThread.java*, *SimControl.java*, *NeighborInfor.java*, *NetInfo.java*, *NodeInfo.java*, *SAXNetworkInfoParser.java*, *SendDelayThread.java*.

1. SimControl.java

This class contains the core of the Central Host functionality and contains a main method. The main method is a sequence of five calls that relate to logical parts of the Simulation Control functionality; *readTopology()*, *neighborhoodSetup()*, *makeNodeConnections()*, *deployNodes()*, *listen4layer2()*.

a. The readTopology Method.

This method uses the *SAXNetworkInfoParser.java* class to parse an XML that is chosen by the user at simulation startup. The network information is then held in a

NetInfo object. This method also calls a private method *chooseFile* that provides a GUI interface for the user to choose the desired XML file to load.

b. The neighborhoodSetup Method.

This method enumerates all the *NodeInfo* objects, held in the *NetInfo* object, using an inner and outer while-loop. This continues until the outer loop enumeration has no more elements. The outer loop enumerates all the nodes once, while the inner loop enumerates all the nodes once for each node of the outer loop enumeration. The outer loop builds the neighborhood for that node while the inner loop provides the next node to evaluate and determine a neighbor relationship. With each permutation of the inner and outer loops of nodes, a comparison is made as to whether this pair of nodes is within detectable range of each other, and if so, it is added to the hashtable within this *NodeInfo* object that contains this node's neighborhood.

c. The makeNodeConnections Method.

This method creates a server socket which to listen for node connections. It utilizes one while-loop to enumerate the *NodeInfo* objects contained in the *NetInfo* object to establish connections with all the nodes of the topology. Once a connection is established with a node, a new *SimCommsThread* is spawned with this socket and the node name. The new thread is then added to a hashtable that will contain reference to all the threads that the *SimControl* uses to communicate with the nodes. This hashtable built with the node name as the key and the communication thread as the object.

d. The deployNodes Method

This method uses a while-loop to enumerate all the *NodeInfo* objects which now contain all the needed information. This information is then passed to the simulated node in a *NODEINFO* type packet through the *forwardPacket* method, which is only used by the *SimControl* since it originates the packet and is not part of the simulated topology. The proper communication thread is chosen by node name.

e. The listen4Layer2 Method

This method listens for UDP packets from a process or application that wishes to connect to the simulation with, at a minimum, layer two functionality. For simplicity of implementation, an empty UDP packet is used. A UDP packet already contains the IP and port of its origination and, therefore, needs no further data to include

this information. This method works with the *Layer2Connect* class in the *UAN.Helpers* package. When this method receives a UDP packet, it is assumed to come from the IP and port listening for UDP packets on the process requesting this connection, from the *Layer2Connect* class. It creates an *Address* object with the IP and port number from the incoming UDP packet. It then calls the *forwardPacket* method with this *Address* object, contained within a *LAYER2* type packet object, to pass this request to the next node with a layer two connection. The simulated node further handles making this connection.

f. Helper Methods

There are several private methods that are also implemented to cleanly divide functionality and ensure code readability. These are: *chooseFile*, *nodeData*, *sendToNeighborhood*, *forwardPacket*, *calcLoss*, and *distance*.

(1). The *chooseFile* Method. This method provides a GUI for the user to choose the XML file containing the node characteristics to load. It utilizes the java file chooser class.

(2). The *nodeData* Method. This method provides the interface for the *SimCommsThread* to pass data from *SimControl* to the simulated node. This method then calls the *sendToNeighborhood* method with the packet passed.

(3) The *sendToNeighborhood* Method. This method enumerates through the hashtable of neighbors of the source node named in the packet. This source node name is retrieved by calling the *getSrcName* method of the *Packet* class. It then forwards this packet to each neighbor, of the source node, via the *SendDelayThread* class.

(4) The *forwardPacket* Method. This method is only utilized during initial setup as all packets, during setup, originate from the SC and does not relate to a neighborhood based on source node. The SC is not in the topology. This allows for directly sending information to the desired node during setup.

(5) The *calcLoss* Method. This method is the implementation of the 'static loss calculations' for this simulation. It takes latitude and longitude of a node pair in decimal degree format and returns the propagation delay in milliseconds after calling the *distance* method to determine the distance between the two entities.

This method must be changed once a different loss calculation is implemented.

(6) The *distance* Method. This method calculates the straight line, cure of the earth, point-to-point distance between the two nodes. It returns this value to the calling method in milliseconds. This millisecond value represents the distance with respect to propagation delay.

g. Testing Methods

There are also four other methods included in the *SimControl* class but will not be discussed. These methods implement different testing and debugging tools and can be studied in the separately provided code if desired.

2. SimCommsThread.java

This class extends the java Thread class and handles the communication between the SC and a simulated node.

`Constructor`	`public SimCommsThread(Socket socket, String name)`
`Methods`	`public void run()`
	`public void dataToNode(Packet p)`
	`private synchronized void dataToSim(Packet p)`

Table 3. SimCommsThread

a. The run Method

The run method is required when extending java class Thread. In this implementation, a while-loop is used to continuously listen for packets from the node. When a packet is received from the node connected to the socket this thread monitors, it is passed to the SC main process via a method a call to method, *dataToSim(packet)*.

b. The dataToSim Method

The synchronized *dataToSim* method passes the packet to the SC via a call of the *nodeData(packet)* in the *SimControl* class. The method is synchronized to ensure safe passing of data, from multiple threads, to the SC. Prior to this, it sets the packet source name to the name of this thread, which is the same as the node name, by calling the *setSrcName* method in the *Packet* class.

c. The dataToNode Method

The *dataToNode(packet)* method is called from the *SimControl* class. This method writes the packet to the object output stream of the socket that was passed to

this thread when this thread was created. The method forces the packet the packet to be forwarded immediately by calling the *flush* method of the output stream.

3. NeighborInfo.java

The *NeighborInfo* object contains a *NodeInfo* object and a delay, in milliseconds as an integer, and is utilized by the *SimControl* class to build a neighborhood relative to each simulated node.

`Constructor`	`public NeighborInfo(NodeInfo node, int delay)`
`Methods`	`public NodeInfo getNode()`
	`public int getDelay()`

Table 4. NeighborInfo

a. The getNode Method

This method returns this node as a *NodeInfo* object.

b. The getDelay Method

This method returns the delay in milliseconds as an integer.

4. NetInfo.java

The NetInfo object contains all the information about this network simulation topology.

`Constructor`	`public NetInfo(String netName)`
`Methods`	`public boolean nodeExists(String name)`
	`public void addNode(NodeInfo node)`
	`public Hashtable getNodes()`
	`public NodeInfo getNodeByName(String name)`

Table 5. NetInfo

a. The nodeExists Method

This method returns a boolean, true or false, with respect to whether a node with the name passed already exists within this topology.

b. The addNode Method

This method adds a node to the *NetInfo* object by creating a new *NodeInfo* object with the passed parameters, and then adding this node to the hash table containing the nodes of this *NetInfo* object.

c. The getNodes Method

This method returns a hashtable of nodes of the *NetInfo* object.

d. The getNodeByName Method

This method returns the *NodeInfo* object with the name passed in the call.

5. NodeInfo.java

The *NodeInfo* object contains all the information about a single simulated node within the network topology.

`Constructor`	`public NodeInfo(String nodeName, String nodeType, String nodeLoc, String nodeDepth, Channel baseChannel, Hashtable secondaryChannels)`
`Methods`	`public Channel getBaseChannel()`
	`public Hashtable getSecondaryChannels()`
	`public String getNodeType()`
	`public String getNodeName()`
	`public String getNodeLoc()`
	`public String getNodeDepth()`
	`public void setNeighbors(Hashtable n)`
	`public Hashtable getNeighbors()`

Table 6. NodeInfo

a. The getBaseChannel Method

This method returns the channel that is the base channel for this node. The base channel is used to transmit but is also able to receive.

b. The getSecondaryChannels Method

This method returns a hashtable of all the channels of this node in excess of the base channel. All of these channels are used to receive.

c. The getNodeType Method

This method returns the node type as a string and can be stationary or mobile. This implementation only uses the stationary type, as mobile nodes are not yet implemented.

d. The setNeighbors Method

This method is called by the *neighborhoodSetup* method of the *SimControl* class, after the neighborhood is built, to update this node's neighborhood hashtable.

e. The getNeighbors Method

This method returns the hashtable containing the neighbors of this node.

6. SAXNetworkInfoParser.java

This parser uses the XML parser provided by java in the org.xml.sax package and implements all methods required by the interface. This implementation is not unique to this work and will not be discussed here but can be found in the provided java documentation and source code if further study is desired. The *endElement* method of this class will be of interest if studied as it is built to handle the specific attributes of this simulation as restricted by the XML DTD provided. If any changes are made to the XML DTD, the *endElement* method will also need to be updated to match.

7. SendDelayThread.java

This class extends the java Thread class and is used to simulate the propagation delay in sending packets between nodes. This method is called from the *SimControl* class with the communication thread in which to use, the packet to send and the amount of delay in milliseconds. This thread then spawns a thread to sleep for the delay time, after which calling the *dataToNode* method in the specific *SimCommsThread* to send the packet. This simulates the packet arriving a the simulated node at the time in which the first bit would have been received in the actual medium.

D. HELPERS PACKAGE

The *UAN.Helpers* package contains classes that do not strictly fit in another package. Most of these classes are utilized by many other classes. This package contains four classes specifically needed for simulation operation; *Address.java*, *Layer2Connect.java*, *Packet.java*, and *Payload.java*. Also included in this package is a ping style class used to test the proper functionality of this simulation, *UAN_Ping.java*.

1. Address.java

This class is used as an object wrapper for an IP and port number to be sent through the simulation. This *Address* object is used by the *SimControl* class to pass the

address information of a process requesting access to a simulated node, chosen by the controller (SC).

`Constructor`	`public Address(InetAddress ip, int port)`
`Methods`	`public InetAddress getIP()`
	`public int getPort()`

Table 7. Address

a. The getIP Method

This method returns the IP address of the process requesting service, as an InetAddress.

b. The getPort Method

This method returns the port of the process requesting service, as an integer. This port is assumed to be a UDP port and is used accordingly by the *NodeControl* and *Layer2Connect* classes.

2. Layer2Connect.java

This class is a helping class for applications to request a connection to the UAN Simulation. The requesting application is expected to include, as a minimum, a layer two process (functionality). This class is offered as an importable class that is to be instantiated by the supported application in order to establish communications with the simulation. It is intended to completely abstract the connection implementation and uses TCP sockets. The supported application exchanges data with the simulation via the *send* and *receive* method call of this helping class.

`Interface`	`public void send(byte[] in)`
	`public byte[] receive()`

Table 8. Layer2Connect Interface

a. The send Interface

This method call takes a byte array and wraps it in the simulation *Payload*, before passing to the simulation. This provides a common interface for any implementation of a layer two process, to use a byte array of data.

b. The receive Interface

This method call returns a byte array to the supported process. It unwraps the array from the *Payload* used in the simulation.

3. Packet.java

This class is the wrapper for all information flow within the simulation. A packet type can consist of *NODEINFO*, *LAYER2*, *SIM*, or *DATA*. The *DATA* type is the only packet type that is used during the simulation run. The other types are used for setup and debugging.

Simulation setup requires a destination to be set for packets originating from the *SimControl* class. Since this is not needed during simulation run, it mandated the use of two constructors as listed below. The first listed is only used during simulation run, while the second listed is used when the *SimControl* class is forwarding information to a specific node.

`Constructors`	`public Packet(int type, Payload payload, int channel)`
	`public Packet(int type, Payload payload, String destination, int channel)`
`Methods`	`public int getChannelID()`
	`public void setSrcName(String s)`
	`public String getSrcName()`
	`public Object getPayload()`
	`public int getPayloadSize()`
	`public void setPayloadSize(int numBytes)`
	`public int getPacketType()`
	`public void setSendTime(long time)`
	`public long getSendTime()`

Table 9. Packet

a. The getChannelID Method

This method returns the unique channel ID as an integer.

b. The setSrcName Method

This method sets the source name of this packet to the name of the node that sent this packet to the *SimControl* class. It is called in the *SimCommsThread* class.

c. The getSrcName Method

This method returns a string of the name of the origination node. This method is called by the *SimControl* class.

d. The getPayload Method

This method returns the packet payload and is called by classes to unwrap packets when needed.

e. The getPayloadSize Method

This method returns the size of the payload contained in this packet. It is called by the *CollisionDetectionThread* class and the *Layer2ThreadTransmit* class.

f. The setPayloadSize Method

This method sets the packets payload size to the length of the byte array passed to the simulation. It is called in the *Layer2ThreadTransmit* class.

g. The getPacketType Method

This method returns a string, naming the packet type. The options are stationary and mobile. Currently only the stationary option is used since mobility is not implemented in this version.

h. The setSendTime Method

This method sets the transmit time of the packet and is called by the *Layer2ThreadTransmit* class.

i. The getSendTime Method

This method returns the transmit time of this packet and is called by the *SendDelayThread* class.

4. Payload.java

This class is the wrapper for data passed down from the layer two process. It consists of several constructors and method that returns the payload as an object. The first constructor is used by the *SimControl* class to pass the node identity to each simulated node. The second constructor is also used by the *SimControl* class, but to send the layer two connection request information to each node. The third constructor is used by the *Layer2Connect* class to wrap the byte array passed from the layer two process.

Constructors	public Payload(NodeInfo node)
	public Payload(Address address)
	public Payload(byte[] in)
Method	public Object getData()

Table 10. Payload

E. NODE PACKAGE

This package contains all classes needed to perform the functionality of the simulated node. These classes include, *CollisionDetectionThread*, *Layer2ThreadReceive*, *Layer2ThreadTransmit*, and *NodeControl*.

1. CollisionDetectionThread.java

This class extends the java Thread class and performs the collision detection logic of transmitting and receiving messages, per channel, per simulated node.

Constructor	public CollisionDetectionThread(boolean base)
Methods	public void run()
	public long check(Packet packet, long transRate)

Table 11. CollisionDetectionThread

a. The run Method

This method, as required when extending the java Thread class, handles the decision process, based on the current state of the collision detection algorithm. This algorithm was defined using a three state FSM in Chapter III. The run method is only part of this algorithm. It tests for the expiration of the collision window and then either drops the packet or passes it to the upper layer, depending on the current state, which is set in the *check* method.

b. The check Method

This method takes the packet passed and determines the time frame of the collision window for that packet's reception. If this is the first packet to arrive, or there is no current collision window, then this window becomes the collision window. If there is a collision window, then it is determined which window will be open longer. The longer window is kept as the collision window and the previous window's packet is dropped since a collision has occurred. If the new packet does not have a longer window, a

collision still occurred and the new packet is dropped, keeping the previous collision window. The packet associated with the previous collision window, which is also considered collided, is dropped once the collision window expires and is accounted for as part of the *run* method described above.

If this channel is a base channel, the transmission window must also be tracked. If there is a transmission window, then it becomes the collision window, the same as previously discussed above with determining collision windows.

2. Layer2ThreadReceive.java

This class is used as half of an implementation for non-blocking I/O using a TCP socket with object streams. This thread is instantiated with the output stream of a socket and only passes data from the simulation to the upper layer.

`Constructor`	`public Layer2ThreadReceive(ObjectOutputStream oos)`
`Methods`	`public void run()`
	`public void send(Payload payload)`

Table 12. Layer2ThreadReceive

a. The run Method

This method is implemented as required when extending the java Thread class and is used only to keep this thread alive.

b. The send Method

This method is called by the *NodeControl* class to pass data to the upper layer by writing to the output stream.

3. Layer2ThreadTransmit.java

This class is the second half of the non-blocking I/O used with *Layer2ThreadReceive* described above.

`Constructor`	`public Layer2ThreadTransmit(ObjectInputStream ois, long transRate)`
`Method`	`public void run()`

Table 13. Layer2ThreadTransmit

a. The run Method

This method is implemented as required when extending the java Thread class and passes data to the simulation from the upper layer by wrapping the *Payload* received from *Layer2Connect* with a *Packet*. It also sets the payload size with a call to *setPayloadSize*, in the *Packet* class, with the length of the byte array that is the payload. It then sleeps for the duration of the transmit window to simulate the reality of only transmitting one message at a time, per transmit channel.

4. NodeControl.java

This class implements the functionality of the simulated node. The main method handles the user I/O and then proceeds into a while-loop to continuously receive data from the simulation. When a packet is received, the packet type is checked and the proper method is called to handle that packet. During initial setup, the *LAYER2* and *NODEINFO* type packets are used. During the simulation run, only the *DATA* type packet is used. An if-else structure is used to check packet type with the DATA condition listed first. This ensures that the simulation is not hindered by nested statements during the run.

When a *DATA* type packet is received, the *collisionDetection* method is called. When a *LAYER2* type packet is received, the *connectLayer2* method is called. When a *NODEINFO* type packet is received, the *buildNode* and then *startCollisionDetectionThreads* methods are called.

a. The collisionDetection Method

This method determines the correct collision detection thread for which to send the packet, based on the ID of the channel on which this packet was transmitted. The *check* method of this thread is then called with this packet.

b. The startCollisionDetectionThreads Method

This method is part of setup and is called after the *buildNode* method has been completed. It starts a *collisionDetectionThread* for each channel of this node.

c. The buildNode Method

This method is part of setup and builds this node with the information sent to it from the *SimControl* class.

d. The getChannelRate Method

This method is called when the transmission rate of a channel is needed. Given the unique channel ID, this method returns the transmission rate.

e. The connectLayer2 Method

This method is part of setup and makes the connection to the layer two process. After receiving the *Address* from the *SimControl* class, it creates a TCP server socket and UDP socket on the same port number. It then sends an empty UDP packet from this port, to the *Address* received. This allows the other side to pull the IP and port number from the UDP packet and use that as the listening port. This "other side" is the *Layer2Connect* class that the layer two process instantiated as a helper function. Once a connection is established, a Layer2ThreadReceive thread and a Layer2ThreadTransmit thread are created with an object output stream and object input stream, respectively.

f. The receiveData Method

This method receives data from the simulation and calls the *send* method of the *Layer2ThreadReceive* class to pass to the upper layer. This method is synchronized to ensure only one collision detection thread can pass data at a time.

g. The transmitData Method

This method is called by the *Layer2ThreadTransmit* class and then writes to the output stream to send data into the simulation.

F. DIRECTION

This chapter has laid out the details, reasons for, the implement of this work. The next chapter will wrap everything up with an overview of the big picture and tool capabilities. It will also define future work options and specific hooks already in place to facilitate specific future works.

V. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

The goal of this thesis was to provide a more hospitable, adaptable, flexible, and easily useable environment in which to test protocols for UANs, as well as providing the ability for the Physics field to test new physical layer devices. This simulation environment was to also provide the glue, or bridge, between the two disciplines by working as a common tool for both.

This thesis met these goals by implementing a simulation tool that provides message distribution between simulated UAN entities, low simulation overhead and proper representation of acoustic characteristics of the medium as enhanced by characteristics of the physical device used to transmit and receive. This tool provides significant flexibility for modeling various protocols representing different layers of the UAN functionality. A key aspect of the model is its focus on reuse, where different protocol implementations or modeling techniques can be instantiated by the user of this work, in order to demonstrate or evaluate the performance of the protocols or techniques of interest without modification to the underlying message exchange mechanism. The only requirement of this underlying mechanism is that data is exchanged via a byte array. This is a reasonable requirement as all programming languages, which might be used with this simulation tool, can handle byte arrays in some form.

This thesis work, in time-base simulations, provides the tool long need for testing high latency protocols required to facilitate underwater acoustic networking. Even though this work does not specifically provide for physical device (modem) testing, it is modularized with forethought to this ability and is ready for implementation in the next version. It is protocol testing where this work is ready for immediate application and use.

With the designed ability to run on a single computer or a network of computers, this thesis provides a tool that is scalable for almost all protocol testing needs from the simplest network topology, to the very complex. Once a topology is defined with a XML

document, the user is assured that each protocol tested, is running with the same network conditions with respect to topology. This specifically allows for affective comparison testing of many different protocols.

The minimizing of simulation overhead was kept to less than 100 milliseconds per RTT. The actual number appears to be closer to 90 milliseconds but enough runs of identical topologies were not completed to evaluate an average and standard deviation that could be reliably subtracted within the simulation. Simulation runs were only performed with intent to validate functionality of the simulation tool itself. The overhead was kept to a minimum by minimizing the busy-wait conditions encountered when using an infinite while-loop. In all cases where a while-loop is used, only one iteration of a loop is needed before a decision is made within this thread or that another thread needs to make a decision based on this thread. To force this utilization, each process using a while-loop is forced to yield at the end of each loop's iteration. This was able to lower simulation overhead by more than 200 milliseconds per RTT.

B. FUTURE WORK

This thesis restricted itself to stationary nodes, but a possible future work is to implement the mobile node. This would require dynamic topology updates to be able to be sent to each node, which would require changes to, at minimum, the *NodeControl* and *SimControl* classes. This ability would greatly increase this tool's value as the ability to simulate UUVs and mobile sensors.

This thesis implemented a static acoustic propagation model. Future work is desired in implementing one of the more complex acoustic models, possibly with the use of a server farm, in order to provide a more dynamic representation of the medium. This would also provide for a more realistic simulation environment in which to test protocols.

This thesis work only allows for topologies to be loaded via a pre-formed XML document. A future work might be implementing a GUI that allows for build a topology from a visual aspect.

This thesis work was restricted to testing protocols, but with forethought of testing physical modem devices also. This interface does not currently exist. This interface would act as a middleware device between the physical modem and the

simulation. This would benefit this work by allowing for physical device testing at the same time with protocols and vice versa.

One other possible future work would be the idea multiple remote UAN agents. That is, a given network may be allocated for part-time use of the UAN simulation and would have a UAN service agent running in the back ground. When a user wants to start a simulation, the Simulation Control agent would then contact and utilize as many waiting node agents a needed for the simulation. This could be accomplished via RPC, RMI or some interface written specifically for the UAN simulation. This would benefit the user greatly in setup and utilization time and possibly the ability to run more than one simulation from the same Simulation Controler.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: SOURCE CODE

A. UAN.SIM PACKAGE

1. SimControl.java

```
package UAN.Sim;

import UAN.Channels.Channel;
import UAN.Helpers.Address;
import UAN.Helpers.Packet;
import UAN.Helpers.Payload;
import UAN.Node.NodeControl;
import java.io.*;
import javax.net.*;
import java.net.*;
import java.util.*;
import java.lang.*;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
/**
 * SimControl is the central point of control and logic for this test-bed
 * environment and must be the first to run.
 * It maintains all socket connections to all clients and employs all the logic
 * for the centralized model view.
 * It also reads network topology from an XML file and then stores this in a
 * <code>NetInfo</code> object.
 */
public class SimControl
{
    /**
     * Used as channel ID if packet is administrative.
     */
    final static int DCID = 99;
    /**
     * This TCP port will listen for connections from remote nodes.
     */
    final static int ADMIN_PORT          = 2875;
    /**
     * This TCP port will listen for connections from remote applications
     * that wish to connect and act as a layer 2 protocol.
     */
    final static int LAYER2_REQUEST_PORT   = 3282;
    /**
     * Defines the TCP Server Socket buffer size for listening.
     */
    final static int BACK_LOG = 100;  //buffer for accepting connections
    /**
     * Defines the maximum distance (time in milliseconds) of realistic
     * communications based on current abilities of 1000 meters and the
     * average speed of sound through water of 1500 meters per second.
     */
    final static int LINK_LOSS_THRESHOLD = 667; //milliseconds
    /**
```

```
     * This hashtable is used as a point for reference to all threads that are
     * spawned for communications between <code>SimControl</code> and the
     * remote nodes.
     */
    static Hashtable threadList     = new Hashtable();
    /**
     * This holds all the information about the network topology read from
     * the XML file.
     */
    static NetInfo myNet            = null;

//**** MAIN ************************************************************
//**********************************************************************
    /**
     * Main contains only the major logic calls.
     * @param args Standard args for Main
     */
    public static void main(String[] args)
    {
        readTopology();
        neighborhoodSetup();
        makeNodeConnections();
        deployNodes();
        listen4layer2();
    }//end main

//******* Body Methods ***************************************************

    /**
     * Parses the XML file into a <code>NetInfo</code> object
     */
    private static void readTopology()
    {
        SAXNetworkInfoParser parser = new SAXNetworkInfoParser();
        myNet = parser.parse(chooseFile());
    }//end readTopology
    /**
     * Build the neighborhoods.
     */
    private static void neighborhoodSetup()
    {
        int delay = 0;
        Enumeration enumOuterLoop = myNet.getNodes().elements();
        while(enumOuterLoop.hasMoreElements())
        {
            Hashtable neighbors = new Hashtable();
            NodeInfo outerLoopNode = (NodeInfo) enumOuterLoop.nextElement();
            Enumeration enumInnerLoop = myNet.getNodes().elements();
            while(enumInnerLoop.hasMoreElements())
            {
                NodeInfo innerLoopNode = (NodeInfo) enumInnerLoop.nextElement();
                if(!outerLoopNode.getNodeName().equals(innerLoopNode.getNodeName()))
                {
                    delay = calcLoss(outerLoopNode, innerLoopNode);
                    if( delay < LINK_LOSS_THRESHOLD)
                    {
```

```
                neighbors.put(innerLoopNode.getNodeName(),
                      new NeighborInfo(innerLoopNode, delay));

                System.out.println("Node: "+
                      outerLoopNode.getNodeName() +
                      " Node: " + innerLoopNode.getNodeName() +
                      " are neighbors with delay: "+delay);
             }//end if less than threshold
           }//end if not myself
        }//end while inner loop
        outerLoopNode.setNeighbors(neighbors);
     }//end while outer loop
  }//end neighborhoodSetup
 /**
  * Spawns a new thread for each node to handle all communications between
  * nodes <code>NodeControl</code> and <code>SimControl</code>.
  * Populates a hashtable with these threads so <code>SimControl</code>
  * can access in the logic section.
  */
  private static void makeNodeConnections()
  {
     NodeInfo thisNode;
     String thisNodeName;
     ServerSocket listenSocket = null;
     Socket socket = null;
     SimCommsThread thisThread;
     Enumeration enumNodes;
     try {
       listenSocket = new ServerSocket(ADMIN_PORT, BACK_LOG);
     } catch (IOException ex) {
        ex.printStackTrace();
        System.out.println("Could not create a ServerSocket");
     }

     System.out.println("Sim Control waiting for connections");
     enumNodes = myNet.getNodes().elements();
     while(enumNodes.hasMoreElements()) //make a connection for each node
     {
        thisNode = (NodeInfo) enumNodes.nextElement();
        thisNodeName = thisNode.getNodeName();
        try {
           socket = listenSocket.accept();
           socket.setKeepAlive(true);
        } catch (IOException ex) {
           ex.printStackTrace();
           System.out.println("Could not create a new Socket");
        }
        System.out.println("DS IP is: "+
                 socket.getLocalAddress().toString());
        thisThread = new SimCommsThread(socket, thisNodeName);
        threadList.put(thisNodeName, thisThread);
        thisThread.start();
        thisThread.setName(thisNodeName);
     }//end while more nodes
     System.out.println("no more nodes to connect to");
  }//end makeNodeConnections
```

```
  /**
   * Sends the node info to each node.  This asigns each remote node
   * connection with the physical properties of one of the nodes described
   * in the topology.
   */
   private static void deployNodes()
   {
      Enumeration enumNodes = myNet.getNodes().elements();
      NodeInfo thisNode;
      Packet newPacket;
      while(enumNodes.hasMoreElements())
      {
         thisNode = (NodeInfo) enumNodes.nextElement();
         newPacket = new Packet(Packet.NODEINFO,
               new Payload(thisNode), thisNode.getNodeName(),DCID);
         forwardPacket(newPacket);
      }
   }//end deployNodes
   /**
    * Receives an empty datagram from an application requesting to connect as
    * a layer 2 protocol and then passes this packet's IP and port number to
    * the next available node in the form of an <code>Address</code> object.
    */
   private static void listen4layer2()
   {
      DatagramSocket dgSkt = null;
      DatagramPacket dgPkt = null;
      int port;
      Enumeration enumNodes;
      NodeInfo thisNode;
      try {
         dgSkt = new DatagramSocket(LAYER2_REQUEST_PORT);
      } catch (SocketException ex) {
         ex.printStackTrace();
         System.out.println("Could not create a new DatagramSocket in listen4Layer2");
      }
      dgPkt = new DatagramPacket(new byte[0], 0);
      enumNodes = myNet.getNodes().elements();
      while(enumNodes.hasMoreElements()) //more nodes to connect layer2
      {
         thisNode = (NodeInfo) enumNodes.nextElement();
         try {
            dgSkt.receive(dgPkt);
         } catch (IOException ex) {
            ex.printStackTrace();
            System.out.println("Problem with receiving datagram packet in listen4Layer2");
         }
         forwardPacket(new Packet(Packet.LAYER2,
                  new Payload(
                    new Address(dgPkt.getAddress(),dgPkt.getPort())),
                       thisNode.getNodeName(), DCID));
      }//end while
   }//end listen4layer2

/****** Helper Methods *****************************************************/
   /**
```

```
 * GUI file chooser for selecting the XML file to load the topology from.
 * @return a <code>File</code> choosen to parse.
 */
private static File chooseFile()
{
   File file = null;
   JFileChooser fileChooser = new JFileChooser();
   fileChooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
   int result = fileChooser.showOpenDialog(new JFrame());
   if (result == JFileChooser.APPROVE_OPTION)
   {
     file = fileChooser.getSelectedFile();
     return file;
   }
   return null;
}//end chooseFile
/**
 * Sends this packet to the node.
 * @param p the <code>Packet</code> being sent.
 */
static void nodeData(Packet p)
{
   sendToNeighborhood(p);
}//end nodeData
/**
 * Emulates the broadcast environment by sending to all of the
 * originating node's neighbors.
 * @param p the <code>Packet</code> being sent.
 */
private static void sendToNeighborhood(Packet p)
{
   Hashtable thisNeighborhood  = myNet.getNodeByName(p.getSrcName()).getNeighbors();
   Enumeration enumNeighbors   = thisNeighborhood.elements();
   SimCommsThread thisThread  = null;
   SendDelayThread pdThread    = null;
   NodeInfo nodeInfo           = null;
   NeighborInfo neighborInfo   = null;
   int delay                   = 0;
   while(enumNeighbors.hasMoreElements())
   {
      neighborInfo = (NeighborInfo) enumNeighbors.nextElement();
      nodeInfo = neighborInfo.getNode();
      delay    = neighborInfo.getDelay();
      thisThread = (SimCommsThread) threadList.get(nodeInfo.getNodeName());
      pdThread = new SendDelayThread(thisThread, p, delay);
      pdThread.start();
   }
}//end sendTodNeighborhood
/**
 * Sends a received packet to the destination node.
 * Used only for configuration and admin.  These packets originated
 * at the sim control.
 * Called by <code>deployNodes</code>
 * @param p the <code>Packet</code> being sent
 */
private static void forwardPacket(Packet p)
```

```
    {
        SimCommsThread thisThread;
        thisThread = (SimCommsThread) threadList.get(p.getDestName());
        thisThread.dataToNode(p);
    }//end forwardPacket
    /**
     * Calculates the loss between nodes based on the distance between
     * nodes return an integer representing the propogation delay between them.
     * @param a any given node
     * @param b any other node to check against
     * @return an <code>Integer</code> representing the propogation delay in milliseconds
     */
    private static int calcLoss(NodeInfo a, NodeInfo b)
    {
        int delay = 0; //milliseconds
        String[] LocA = a.getNodeLoc().split(" ");
        String[] LocB = b.getNodeLoc().split(" ");
        double LatA = Double.parseDouble(LocA[0]);
        double LonA = Double.parseDouble(LocA[1]);
        double LatB = Double.parseDouble(LocB[0]);
        double LonB = Double.parseDouble(LocB[1]);
        double dist = distance(LatA, LonA, LatB, LonB);
        System.out.println("Node pair "+a.getNodeName()+"::"+
                b.getNodeName()+" has distance: "+dist+" meters");
        //will check if loss of precision makes a difference
        //may change to nanoseconds later
        delay = (int)(dist / 1.5);
        //1.5 comes from dividing by 1500 m/s and multipling by 1000 to get milliseconds
        return delay;
    }//end calcLoss
    /**
     * Calculates distance in meters between two points given as latitude and
     * longitude in decimal degree format.
     * @param lat1 the latitude of the first node
     * @param lon1 the longitude of the first node
     * @param lat2 the latitude of the second node
     * @param lon2 the longitude of the second node
     * @return a <code>double</code> representation of the distance between nodes in meters.
     */
    private static double distance(double lat1, double lon1, double lat2, double lon2)
    {
      double theta = lon1 - lon2;
      double dist = Math.sin(Math.toRadians(lat1)) * Math.sin(Math.toRadians(lat2)) +
                    Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) *
                    Math.cos(Math.toRadians(theta));
      dist = Math.toDegrees(Math.acos(dist));
      dist = dist * 60; //nautical miles
      dist = dist * 1852; //meters
      return dist;
    }//end distance

/** Testing Methods ********************************************************/
    /**
     * Testing method to display nodes and modems of the the whole
     * topology.
     * @param net the whole network topology.
```

```
 */
private static void display(NetInfo net)
{
   Enumeration eNode = net.getNodes().elements();
   while(eNode.hasMoreElements())
   {
      NodeInfo n = (NodeInfo) eNode.nextElement();
      System.out.println("Node: " + n.getNodeName());
      Enumeration eChannels = n.getSecondaryChannels().elements();
      while(eChannels.hasMoreElements())
      {
         Channel m = (Channel) eChannels.nextElement();
         System.out.println("Channel: " + m.getName());
      }
   }
}//end display
/**
 * Testing method
 * Receives packet object from a node and then sends to other node(s) based
 * on destination.
 * @param p the packet to send.
 */
public static void nodeData1(Packet p)
{
   if(p.getDestName().equalsIgnoreCase(NodeControl.BROADCAST_ID))
   {
      broadcastNodes(p);
   }
   else
   {
      forwardPacket(p);
   }
}//end nodeData1
/**
 * Sends a received packet to all nodes when the destination ID was set to
 * Broadcast.
 * This is a testing method only and does not limit sending to a
 * neighborhood.
 * @param p the <code>Packet</code>
 */
private static void broadcastNodes(Packet p)
{
   SimCommsThread thisThread;
   Enumeration thList = threadList.elements();
   while(thList.hasMoreElements())
   {
      thisThread = (SimCommsThread) thList.nextElement();
      thisThread.dataToNode(p);
   }//end while
}//end broadcastNodes
/**
 * testing method call to distance
 */
private static void calcDistance()
{
   System.out.println(distance(32, -96, 32, -96.02) + " Meters\n");
```

```
    }//end calcDistance
}//end SimStaion
```

2. SimCommsThread.java

```
package UAN.Sim;

import UAN.Helpers.Packet;
import UAN.Helpers.Payload;
import java.io.*;
import java.net.*;
import java.lang.*;
/**
 * <code>SimCommsThread</code> extends <code>Thread</code> and handles
 * all communication between <code>SimControl</code> and each node.
 * A seperate thread is used to communicate to each instance of
 * <code>NodeControl</code> running in the simulation.
 */
public class SimCommsThread extends Thread
{
    private Socket skt = null;
    private ObjectOutputStream oos;
    private ObjectInputStream ois;
    /**
     * Constuctor
     * @param socket the <code>Socket</code> that this thread manages.
     * @param name the <code>String</code> that represents this thread's name.
     */
    public SimCommsThread(Socket socket, String name)
    {
        super(name);
        skt = socket;
    }//end constructor
    /**
     * Thread implementation of run.
     */
    public void run()
    {
        try
        {
            oos = new ObjectOutputStream(skt.getOutputStream());
            ois = new ObjectInputStream(skt.getInputStream());
            Payload s;
            Packet p;
            while(true)
            {
                p = (Packet) ois.readObject();
                dataToSim(p);
                yield(); //minimizing busy wait
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
                ex.printStackTrace();
        }//end trycatch
```

```
    }//end run
    /**
     * Sends a packet from <code>SimControl</code> to node via the
     * output stream.
     *
     * @param p the <code>Packet</code> being sent.
     */
    public void dataToNode(Packet p)
    {
        try
        {
            oos.writeObject(p);
            oos.flush();

        } catch (IOException ex) {
            ex.printStackTrace();
            System.out.println("Could not write to Output Stream in dataToNode");
        }
    }//end dataToNode
    /**
     * A <code>Packet</code> was received from a node in <code>run</code>
     * and is sent to <code>SimControl</code>.
     * Synchronized to ensure only one thread is passing data to
     * <code>SimControl</code> at a time.
     *
     * @param p the <code>Packet</code> being sent.
     */
    private synchronized void dataToSim(Packet p)
    {
        if(p.getPacketType() == Packet.SIM)
        {
            SimControl.nodeData1(p);
        }
        else
        {
            p.setSrcName(this.getName());
            SimControl.nodeData(p);
        }
    }//end dataToSim
}//end SimCommsThread
```

3. NeighborInfo.java

```
package UAN.Sim;

import java.io.Serializable;
/**
 * Holds the different delay costs for the same node wrt different
 * neighborhoods.  This is also the place later to store the loss data from
 * the server farm about the acoustic channel.  This supports bidirectional
 * differences in transmission characteristics.
 */
public class NeighborInfo implements Serializable
{
    /**
     * Holds the node information
```

```
     */
    private NodeInfo nodeInfo;
    /**
     * Holds the delay in milliseconds
     */
    private int delay;
    /**
     * Constructor
     * @param node the <code>NodeInfo</code>
     * @param num this node's propogation delay relative to the node that this
     * node a neighbor of.
     */
    public NeighborInfo(NodeInfo node, int delay)
    {
       nodeInfo    = node;
       this.delay  = delay;
    }
    /**
     * Use this method to get the Node
     * @return <code>NodeInfo</code>
     */
    public NodeInfo getNode()
    {
       return nodeInfo;
    }
    /**
     * Use this method to get the delay time
     * @return <code>Integer</code> representing delay time
     */
    public int getDelay()
    {
       return delay;
    }

}//end NeighborNode
```

4. NetInfo.java

```
package UAN.Sim;

import UAN.Channels.Channel;
import UAN.Sim.NodeInfo;
import java.util.*;
/**
 * A record of information that describes a network.
 */
public class NetInfo
{
    /**
     * This network's name.
     */
    private String netName;
    /**
     * Hashtable to store node info of this topology.
     */
```

```
  private Hashtable nodeTable = new Hashtable();//store all the nodes in this net
 /**
  * Set the network name.
  * @param netName a <code>String</code> of the network name.
  */
 public NetInfo(String netName)
 {
  this.netName =  netName;
 }
/**
  * Checks whether the node already exists on the network.
  * @param name the <code>String</code> of the node name.
  * @return <code>boolean</code> representing whether the node already exists.
  */
 public boolean nodeExists(String name)
 {
  if (nodeTable.get(name) == null)
  {
   return false;
  }
  return true;
 } //end of nodeExists()
  /**
   * Adds a node to this NetInfo object.
   * @param nodeName a <code>String</code> of node name.
   * @param nodeType a <code>String</code> of node type.
   * @param nodeLoc a <code>String</code> of node location.
   * @param nodeDepth a <code>String</code> of node depth.
   * @param baseChannel a <code>Channel</code> for node base channel.
   * @param secondaryChannelTable a <code>Hashtable</code> of secondary channels.
   * @return the new node as <code>NodeInfo</code>
   */
 public NodeInfo addNode(String nodeName,
                         String nodeType,
                         String nodeLoc,
                         String nodeDepth,
                         Channel baseChannel,
                         Hashtable secondaryChannelTable)
 {
  NodeInfo newNode = new NodeInfo( nodeName,
                                   nodeType,
                                   nodeLoc,
                                   nodeDepth,
                                   baseChannel,
                                   secondaryChannelTable);

  //add NodeInfo object to hash table
  nodeTable.put(nodeName, newNode);
  return newNode;
 } //end of addNode()
 /**
  * Node table of this NetInfo object.
  * @return Table of nodes as <code>Hashtable</code>
  */
 public Hashtable getNodes()
 {
```

```
    return nodeTable;
  }
  /**
   * Returns the node belonging to the network, and whose name matches
   * with the given name.  If no node can be found, a null is returned.
   * @param name the <code>String</code> of node name.
   * @return <code>NodeInfo</code> of the node name.
   */
  public NodeInfo getNodeByName(String name)
  {
    return (NodeInfo) nodeTable.get(name);
  }
}//end of class NetInfo
```

5. NodeInfo.java

```
package UAN.Sim;

import UAN.Channels.*;
import java.io.Serializable;
import java.net.InetAddress;
import java.util.*;

/**
 * Holds the information about a given node in the topology.  Is serializable so
 * that it can be sent as an object over input and output streams.
 */
public class NodeInfo implements Serializable
{
  /**
   * A <code>String</code> for this node's name.
   */
  private String nodeName;
  /**
   * A <code>String</code> for this node's type.
   */
  private String nodeType;
  /**
   * A <code>String</code> for this node's location.
   */
  private String nodeLoc;
  /**
   * A <code>String</code> for this node's depth in lat/long, decimal degree format.
   */
  private String nodeDepth;
  /**
   * An <code>InetAddress</code> for the physical on which this node resides.
   */
  private InetAddress hostIPv4;
  /**
   * A <code>short</code> to represent the TCP port.
   */
  private short tcpPortNumber;
  /**
   * Base channel will always transmit
```

```
     */
    private Channel baseChannel;
    /**
     * A <code>ArrayList</code> of channels.
     */
    private Hashtable secondaryChannels;
    /**
     * A <code>Hashtable</code> of neighbors.
     */
    private Hashtable neighbors;

    /**
      * Constructs a NodeInfo object.
      * @param nodeName the <code>String</code> of the node name.
      * @param nodeType the <code>String</code> of the node type.
      * @param nodeLoc the <code>String</code> of the node location.
      * @param nodeDepth the <code>String</code> of the node depth.
      * @param baseChannel the <code>Channel</code> that is the base channel. It can transmit
and receive.
      * @param secondaryChannels a <code>Hashtable</code> of other channels of this node.
      */
    public NodeInfo(String nodeName,
                    String nodeType,
                    String nodeLoc,
                    String nodeDepth,
                    Channel baseChannel,
                    Hashtable secondaryChannels)
    {
     this.nodeName = nodeName;
     this.nodeType = nodeType;
     this.nodeLoc = nodeLoc;
     this.nodeDepth = nodeDepth;
     this.baseChannel = baseChannel;
     this.secondaryChannels = secondaryChannels;
    } //end of constructor
    /**
     * Returns the base channel of this node
     * @return Channel
     */
    public Channel getBaseChannel()
    {
       return baseChannel;
    }
    /**
     * Returns the secondary channel hashtable of this Node.
     * @return <code>Hashtable</code>
     */
    public Hashtable getSecondaryChannels()
    {
     return secondaryChannels;
    }
    /**
     * Returns the node type of this NodeInfo object.
     * @return <code>String</code> for Node type
     */
    public String getNodeType()
```

```
  {
   return nodeType;
  }
  /**
   * Returns the node name of this NodeInfo object.
   * @return Name of the node as a <code>String</code>.
   */
  public String getNodeName()
  {
   return nodeName;
  }
  /**
   * Gets this node's location
   * @return this node's location as a <code>String</code>
   */
  public String getNodeLoc()
  {
     return nodeLoc;
  }
  /**
   * Returns the water depth of this node
   * @return the water depth of this node
   */
  public String getNodeDepth()
  {
     return nodeDepth;
  }
  /**
   * Set the neighborhood for this node.
   * @param n the <code>Hashtable</code> of neighbors.
   */
  public void setNeighbors(Hashtable n)
  {
     neighbors = n;
  }
  /**
   * Returns the neighborhood.
   * @return <code>Hashtable</code> of neighbors.
   */
  public Hashtable getNeighbors()
  {
     return neighbors;
  }
}//end of class NodeInfo
```

6. SAXNetworkInfoParser.java

```
package UAN.Sim;

import java.io.*;
import java.net.*;
import java.util.*;

import UAN.Channels.*;
```

```
//XML-SAX parser imports
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
/**
 * Parses an XML file using SAX parser to load the network information of the
 * <code>SimControl</code>.
 */
public class SAXNetworkInfoParser
{
  NetInfo myNet = new NetInfo("UAN");
  /**
   * Default constructor
   */
  public SAXNetworkInfoParser()
  {
     //empty
  }
  /**
   * Parses an XML file using the SAX parser.
   *
   * @param file Input XML file
   * @return <code>NetInfo</code> containing the newly parsed information.
   */
  public NetInfo parse (File file)
  {
   URL url = null;
   try
   {
    url = file.toURL();
   }
   catch (MalformedURLException e)
   {
    System.out.println("Cannot load file.  Malformed URL: " + e.getMessage());
    return null;
   }
   ContentHandler contentHandler = new UANTopologyContentHandler();
   ErrorHandler   errorHandler   = new MyErrorHandler();
   try
   {
    //Instantiate a SAX parser
    XMLReader                                                parser                                =
XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
    //Register the content handler
    parser.setContentHandler(contentHandler);
    //Register the error handler
    parser.setErrorHandler(errorHandler);
    //Parse the document
    parser.parse(url.toString());
   }
   catch (IOException ie)
```

```
  {
    System.out.println("Cannot load file.  Error reading URL: " + ie.getMessage());
  }
  catch (SAXException se)
  {
    System.out.println("Error in parsing: " + se.getMessage());
    se.printStackTrace(System.err);
  }
  return myNet;
} //end of parse()
/**
 * Inner class <code>UANTopologyContentHandler</code> implements the SAX
 * interface and parses the document based on the events that happen inside
 * the document.  It establishes the basic data structures that gather the
 * network topology information from the XML file.
 */
class UANTopologyContentHandler implements ContentHandler
{
  //Hold onto the locator for location information
  private Locator locator;
  private Channel newChannel = null;
  private Channel baseChannel = null;
  private boolean BASE_SET = false;

  //global & temp variables to store values read from the XML file
  String element     = "";
  String value       = "";
  String hostIPv4    = "";
  String nodeName    = "";
  String nodeType    = "";
  String nodeLocation = "";
  String nodeDepth   = "";
  String channelName  = "";
  String channelID  = "";

  NodeInfo newNode;

  Hashtable secChannelTable;
  /**
   * This method gives the capability to define the exact location while
   * parsing the XML file.
   * @param locator Document locator
   */
  public void setDocumentLocator(Locator locator)
  {
    System.out.println("    * setDocumentLocator() called");
    //We save this for later use if desired.
    this.locator = locator;
  }
  /**
   * In this method there can be any kind of statements that we would like
   * to occur when we first open an XML document.
   * @throws SAXException when thins go wrong
   */
  public void startDocument() throws SAXException
  {
```

```
  System.out.println("Parsing begins...");
}
/**
 * This reports the occurrence of an actual element.  It will include the
 * element's attributes, with the exception of XML vocabulary specific
 * attributes like "DTD", ...
 * @param namespaceURI Namespace URI
 * @param localName Local name
 * @param rawName Raw name
 * @param atts Attributes
 * @throws SAXException when things go wrong
 */
public void startElement(String namespaceURI,
        String localName, String rawName, Attributes atts)
        throws SAXException
{
  System.out.println(" startElement: OK right here. " + localName);
  element = localName;
}
/**
 * Returns the real value stored in the XML element and then converted
 * to a string " array of characters ".
 * @param ch Characters
 * @param start Start position
 * @param end End position
 * @throws SAXException when things go wrong
 */
public void characters(char [] ch, int start, int end) throws SAXException
{
  String s = new String(ch, start, end);
  value = s;
}
/**
 * Indicates the end of an element is reached.  Note that the parser does
 * not distinguish between empty elements and non-empty elements, so this
 * will occur uniformly.  We gather the value of the element when we reach
 * the end of the element.
 * @param namespaceURI Namespace URI
 * @param localName Local name
 * @param rawName Raw name
 * @throws SAXException when things go wrong
 */
public void endElement(String namespaceURI,
        String localName, String rawName) throws SAXException
{
  if (localName.equals("UAN_Net"))
  {
    System.out.println("This completes a UAN network!!");
  }
  else if (localName.equals("Node"))
  {
    System.out.println("============= End of a new node =============");
    //Check if node already exist.  If it indeed exists already, warn user
    if (myNet.nodeExists(nodeName))
    {
      System.out.println("Node already exists: " + nodeName +
```

```
                    " node not added to the network");
    return;
   }
   //Check if node has any channel.  If it doesn't, warn user
   if (baseChannel == null)
   {
    System.out.println("Node does not have any channel: " + nodeName +
                    " node not added to the network");
    return;
   }
      newNode = myNet.addNode( nodeName,
                        nodeType,
                        nodeLocation,
                        nodeDepth,
                        baseChannel,
                        secChannelTable);
}
else if (localName.equals("NodeType"))
{
  nodeType = value;
  System.out.println(" Node Type = " + nodeType);
}
else if (localName.equals("NodeLocation"))
{
    nodeLocation = value;
    System.out.println(" Node Location = " + nodeLocation);
}
else if (localName.equals("NodeDepth"))
{
    nodeDepth = value;
    System.out.println(" Node Depth = " + nodeDepth);
}
else if (localName.equals("NodeName"))
{
  nodeName = value;
  System.out.println("============= Start of a new node ============");
  System.out.println(" Node Name = " + nodeName);
  BASE_SET = false;

 //This is the beginning of a new node, initialize the following variables
  secChannelTable = new Hashtable();
}
else if (localName.equals("Channel"))
{
  System.out.println("============= End of New Channel ==============");
}
else if (localName.equals("ChannelName"))
{
    channelName = value;
    System.out.println(" Channel Name = " + channelName);
}
else if (localName.equals("ChannelID"))
{
    channelID = value;
    System.out.println(" Channel Type = " + channelID);
    //Create a new channel record.  Since the node object has not been
```

```
     //created yet, we use null as the parent node for now.  The parent node will
     //be set once the node information has been parsed completely.
    if(channelID.equals("0"))
    {
      newChannel = new ChannelZero(channelName);
      System.out.println("This is a Channel Zero");
    }
    else if(channelID.equals("1"))
    {
       newChannel = new ChannelOne(channelName);
       System.out.println("This is a Channel One");
    }
    else if(channelID.equals("2"))
    {
       newChannel = new ChannelTwo(channelName);
       System.out.println("This is a Channel Two");
    }
    if(!BASE_SET)
    {
       baseChannel = newChannel;
       BASE_SET = true;
       System.out.println("Base Channel set as " + baseChannel.getName());
    }
    else
    {
      secChannelTable.put(channelName, newChannel);
    }
  }
} //end of endElement()
/**
 * Indicates the end of the XML document.  It is reached when it finishes
 * all the parsing events inside that we would like to occur when we first
 * open an XML document.
 * @throws SAXException when things go wrong
 */
public void endDocument() throws SAXException
{
  System.out.println("...Parsing ends.");
} // end of endDocument()
/**
 * This method is used when processing instructions (PI) are found in
 * the XML file.  The current XML file doesn't support PI.
 * @param target
 * @param data
 * @throws SAXException when things go wrong
 */
public void processingInstruction (String target,
       String data) throws SAXException
{
  System.out.println("PI: Target: " + target + " and Data: " + data);
}
/**
 * This method is used in case XML name spaces are used.
 * In the current use of XML, name spaces are not supported.
 * @param prefix
 * @param uri
```

```
 */
 public void startPrefixMapping (String prefix, String uri)
 {
  System.out.println("Mapping starts for prefix "+
        prefix + " mapped to URI " + uri);
 }
 /**
  * This method is used in case XML name spaces are used.
  * In the current use of XML, name spaces are not supported.
  * @param prefix
  */
 public void endPrefixMapping (String prefix)
 {
  System.out.println("Mappine ends for prefix " + prefix);
 }
 /**
  * This method provides the ability to report white spaces.
  * @param  ch Character array
  * @param  start Start position
  * @param  end End position
  * @throws SAXException when things go wrong
  */
 public void ignorableWhitespace (char [] ch, int start, int end)
    throws SAXException
 {
  String s = new String(ch, start, end);
  System.out.println("ignorableWhitespace: [" + s + "]");
 }

 /**
  * This method will report an entity that is skipped by the parser.
  * This should only occur for non-validating parsers, and then is still
  * implementation dependent behavior.
  * @param name Entity name
  * @throws SAXException when things go wrong
  */
 public void skippedEntity (String name) throws SAXException
 {
  System.out.println("Skipping entity " + name);
 }
} //end of inner class UAN NetworkTopologyContentHandler
/**
 * Inner class <em>MyErrorHandler</em> implements the SAX
 * ErrorHandler interface.
 */
class MyErrorHandler implements ErrorHandler
{
 /**
  * Reports a warning that has occurred; this indicates that while
  * no XML rules were "broken", something appears to be incorrect or
  * missing an entity that is skipped by the parser.
  * @param  exception Parser exception
  * @throws SAXException
  */
 public void warning (SAXParseException exception) throws SAXException
 {
```

```
    System.out.println("**Parsing Warning**\n" +
                       "  Line:     " + exception.getLineNumber() + "\n" +
                       "  URI:      " + exception.getSystemId() + "\n" +
                       "  Message:  " + exception.getMessage());
    throw new SAXException("Warning encountered");
   }
   /**
    * Reports an error if a non-critical parsing error has occurred.
    * This error indicates that even if an XML rules was broken, the
    * parsing can continue.
    * @param  exception Parse exception
    * @throws SAXException
    */
   public void error (SAXParseException exception) throws SAXException
   {
    System.out.println("**Parsing Error**\n" +
                       "  Line:     " + exception.getLineNumber() + "\n" +
                       "  URI:      " + exception.getSystemId() + "\n" +
                       "  Message:  " + exception.getMessage());
    throw new SAXException("Error encountered");
   }
   /**
    * Reports a fatal error if a critical parsing error has occurred.  This
    * fatal error indicates that the parsing process can't continue because
    * of a major violation to XML rules.
    * @param  exception Parse exception
    * @throws SAXException
    */
   public void fatalError (SAXParseException exception) throws SAXException
   {
    System.out.println("**Parsing Fatal Error**\n" +
                       "  Line:     " + exception.getLineNumber() + "\n" +
                       "  URI:      " + exception.getSystemId() + "\n" +
                       "  Message:  " + exception.getMessage());
    throw new SAXException("Fatal Error encountered");
   }
  }//end of inner class MyErrorHandler
}//end of class SAXNetworInfoParser
```

7. SendDelayThread.java

```
package UAN.Sim;

import UAN.Helpers.Packet;
import java.io.IOException;
/**
 * This thread delays the sending of the <code>Packet</code> until the first
 * bit would have arrived based on the propogation delay.  Uses the java sleep
 * system call to implement the delay.
 */
public class SendDelayThread extends Thread
{
   SimCommsThread commsThread;
   Packet packet;
   int delay;
```

```
    long temp;
    /**
     * Constructor
     *
     * @param t the <code>SimCommsThread</code> to send to.
     * @param p the <code>Packet</code> to be sent.
     * @param d the <code>Integer</code> representing the delay in milliseconds.
     */
    public SendDelayThread(SimCommsThread t, Packet p, int d)
    {
        commsThread = t;
        packet      = p;
        delay       = d;
    }
    /**
     * Thread implementation for run
     */
    public void run()
    {
      try
        {
            System.out.println("Delay "+delay+
                    "ms before sending packet from "+
                    packet.getSrcName());
            temp = System.currentTimeMillis() - packet.getSendTime();
            Thread.sleep(delay - (int)temp);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        commsThread.dataToNode(packet);
    }
}
```

B. UAN.NODE PACKAGE

1. NodeControl.java

```
package UAN.Node;

import UAN.Channels.Channel;
import UAN.Sim.NodeInfo;
import UAN.Helpers.Address;
import UAN.Helpers.Packet;
import UAN.Helpers.Payload;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import java.util.*;
import javax.swing.JFrame;
/**
 * Handles all aspects of the simulation node.
 */
public class NodeControl
{
    static final boolean BASE = true;
```

```
/**
 * used as a monitor for transmit and CD threads.  No specifier to restrict
 * access to this package.
 */
static boolean receivingBase = false;
/**
 * used as a monitor for transmit and CD threads.  No specifier to restrict
 * access to this package.
 */
static boolean transmitingBase = false;
/**
 * used for testing demonstration only
 */
public final static String BROADCAST_ID = "ALL";
/**
 * TCP port number for nodes to connect to <code>SimControl</code>
 */
private static int adminPort = 2875;
/**
 * The IP address of <code>SimControl</code>
 */
private static String simIP = "192.168.1.1"; //default
/**
 * Communication from <code>SimControl</code> to <code>NodeControl</code>.
 */
private static ObjectInputStream oisSim;
/**
 * Communication from <code>NodeControl</code> to <code>SimControl</code>.
 */
private static ObjectOutputStream oosSim;
/**
 * This node's info
 */
private static NodeInfo nodeInfo = null;
/**
 * Handles communication from node to layer 2 protocol.
 */
private static Layer2ThreadReceive receiveThread;
/**
 * Handles communication from layer 2 protocol to node.
 */
private static Layer2ThreadTransmit transmitThread;
/**
 * The transmission rate of the base channel.
 */
private static long baseTransmitRate;
/**
 * The unique ID associated with the base channel
 */
static int baseChannelID;
/**
 * The user name of this base channel
 */
private static String baseChannelName;
/**
 * This node's base channel
```

```
 */
private static Channel baseChannel = null;
/**
 * This node's secondary channels
 */
private static Channel[] secondaryChannels       = new Channel[10];
/**
 * The hashtable to contain the thread list of the collision detection
 * threads running for each channel
 */
private static Hashtable collisionThreadList    = new Hashtable();
/**
 * A reference for a collision detection thread
 */
private static CollisionDetectionThread cdThread;
/**
 * Main
 * @param args is args
 */
public static void main(String[] args)
{
    Socket simSocket = null;
    Payload payload = null;
    Packet packet;
    int packetType;
    String tempPort = null;

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("\nEnter the Sim Control Host IP address <ENTER>::default");
    try {
        simIP = br.readLine().trim();
        if(simIP.length()<7)
        {
            simIP = "192.168.1.1";
        }
        System.out.println("\nEnter the Sim Control Host TCP port number <ENTER>::default");
        tempPort = br.readLine().trim();
    }
    catch (IOException ex)
    {
        System.out.println("User input invalid");
    }
    try {
        adminPort = Integer.parseInt(tempPort);
    }
    catch(Exception e)
    {
        //do nothing, using default port number
        System.out.println("Using default port number");
    }
    try {
        //build I/O :: output must be first
        simSocket = new Socket(simIP, adminPort);
    } catch (UnknownHostException ex) {
        ex.printStackTrace();
    } catch (IOException ex) {
```

```
            ex.printStackTrace();
        }
        try {
            oosSim = new ObjectOutputStream(simSocket.getOutputStream());
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        try {
            oisSim = new ObjectInputStream(simSocket.getInputStream());
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        System.out.println("Node connected to IP "+simIP+" and port "+
                adminPort+" from IP "+simSocket.getLocalAddress()+ " on port "+
                simSocket.getLocalPort());
        try {
            while((packet = (Packet) oisSim.readObject()) != null)
            {
                payload     = (Payload) packet.getPayload();
                packetType  = packet.getPacketType();
                Thread cdThread;

                if(packetType == Packet.DATA)
                {
                    collisionDetection(packet);
                }
                else if(packetType == Packet.LAYER2)
                {
                    Address address = (Address)payload.getData();
                    connectLayer2(address);
                }
                else if(packetType == Packet.NODEINFO)
                {
                    nodeInfo = (NodeInfo)payload.getData();
                    try{
                        buildNode();
                    }catch (Exception e){
                        e.printStackTrace();
                    }
                    startCollisionDetectionThreads();
                }
                else if(packetType == Packet.SIM)
                {
                    if(payload.getData().toString().trim().equals("Quit"))
                    {
                        System.exit(0);
                    }
                }
            }//end while
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }//end while
    }//end main
```

```
/******** Methods ****************************************************/
   /**
    * Gives this packet to the collison detection thread for the channel
    * from which the originating node transmitted.
    */
   private static void collisionDetection(Packet packet)
   {
      Channel thisChannel;
      long duration;
      if(packet.getChannelID() == baseChannelID)
      {
         cdThread = (CollisionDetectionThread) collisionThreadList.get(baseChannelName);
         duration = cdThread.check(packet, baseTransmitRate);
      }
      else
      {
         thisChannel = (Channel) secondaryChannels[packet.getChannelID()];
         cdThread = (CollisionDetectionThread) collisionThreadList.get(thisChannel.getName());
         duration = cdThread.check(packet, getChannelRate(packet.getChannelID()));
      }
   }
   /**
    * Starts a seperate collison dectection thread for each channel.
    */
   private static void startCollisionDetectionThreads()
   {
      CollisionDetectionThread myThread = new CollisionDetectionThread(BASE);
      myThread.start();
      collisionThreadList.put(baseChannelName, myThread);

      Enumeration enumSecChannels = nodeInfo.getSecondaryChannels().elements();
      Channel thisChannel;
      while(enumSecChannels.hasMoreElements())
      {
         myThread = new CollisionDetectionThread(!BASE);
         myThread.start();
         thisChannel = (Channel) enumSecChannels.nextElement();
         collisionThreadList.put(thisChannel.getName(), myThread);
      }
   }
   /**
    * Builds this node from the nodeInfo object
    */
   private static void buildNode()
   {
      baseChannel = nodeInfo.getBaseChannel();
      baseChannelID = baseChannel.getID();
      baseChannelName = baseChannel.getName();
      baseTransmitRate = baseChannel.getTransmitRate();
      Enumeration sc = nodeInfo.getSecondaryChannels().elements();
      Channel thisChannel;
      while(sc.hasMoreElements())
      {
         thisChannel = (Channel) sc.nextElement();
         secondaryChannels[thisChannel.getID()] = thisChannel;
      }
```

```
            System.out.println("This Node's name is "+nodeInfo.getNodeName());
        }//end buildNode
        /**
         * Returns the transmit rate of a channel with this ID
         * @return the transmit rate
         */
        private static long getChannelRate(int ID)
        {
            Channel channel = (Channel) secondaryChannels[ID];
            return channel.getTransmitRate();
        }//end getChannelRate
        /**
         * Connects to the address received and then spawns a thread to handle the
         * output stream and a seperate thread to handle the input stream.  Output
         * and input are relative to layer1's relation to layer2.
         * @param address the <code>Address</code> of the layer 2 protocol host
         */
        private static void connectLayer2(Address address)
        {
            //address from layer2 request
            InetAddress layer2IP = address.getIP();
            int layer2Port = address.getPort();
            DatagramSocket dgSkt    = null;
            DatagramPacket dgPkt    = null;
            ServerSocket ss = null;
            Socket skt      = null;
            ObjectInputStream oisL2 = null;
            ObjectOutputStream oosL2 = null;
            try {
                ss = new ServerSocket(0);
            } catch (IOException ex) {
                ex.printStackTrace();
                System.out.println("Could not make new Server Socket in connectLayer2");
            }
            try {

                dgSkt = new DatagramSocket(ss.getLocalPort());
            } catch (SocketException sx) {
                sx.printStackTrace();
                System.out.println("Could not make the datagram socket on the tcp port in
connectLayer2");
            }
            dgPkt = new DatagramPacket(new byte[0], 0, layer2IP, layer2Port);
            try {
                dgSkt.send(dgPkt);
            } catch(IOException ex) {
                ex.printStackTrace();
                System.out.println("Could not send Datagram to layer2 App in connectLayer2");
            }
            try {
                skt = ss.accept();
            } catch (IOException ex) {
                ex.printStackTrace();
                System.out.println("Could not make connection from Server Socket in connectLayer2");
            }
            try {
```

```
            oisL2 = new ObjectInputStream(skt.getInputStream());
        } catch (IOException ex) {
            ex.printStackTrace();
            System.out.println("Could not get input stream in connectLayer2");
        }
        try {
            oosL2 = new ObjectOutputStream(skt.getOutputStream());
        } catch (IOException ex) {
            ex.printStackTrace();
            System.out.println("Could not get output stream in connectLayer2");
        }
        receiveThread = new Layer2ThreadReceive(oosL2);
        transmitThread = new Layer2ThreadTransmit(oisL2, baseTransmitRate);
        receiveThread.start();
        transmitThread.start();
    }//end connectLayer2
    /**
     * Sends packet to layer 2 thread.
     * @param packet the <code>Packet</code> to being sent.
     */
    public synchronized static void receiveData(Packet packet)
    {
        receiveThread.send((Payload) packet.getPayload());
    }//end receiveData
    /**
     * Called from <code>Layer2ThreadTransmit</code> to pass data from layer2
     * to this simulated physical layer.
     * @param p the <code>Packet</code> to being sent.
     */
    public static void transmitData(Packet p)
    {
        try {
            oosSim.writeObject(p);
            oosSim.flush();
        } catch (IOException ex) {
            ex.printStackTrace();
            System.out.println("Unable to write to output stream in transmitData");
        }
    }//end transmitData
}//end NodeControl
```

2. CollisionDetectionThread.java

```
package UAN.Node;

import UAN.Helpers.Packet;
/**
 * Handles the collision detection logic for all packets received by a given
 * node.
 */
public class CollisionDetectionThread extends Thread
{
    //state can be 0, 5, or 7 respresentation of three bits
    // [busy, collision, message]
    //state 0 is no message in que for decission
```

```
//state 5 is message in wait with no collision yet
//state 7 is message in wait and collision has accured
//state 9 is waiting on a transmission window to expire
final static int IDLE = 0;
final static int BUSY = 5;
final static int COLL = 7;
final static int TRAN = 9;

int state;
long collisionWindow, tempTime, duration; //milliseconds
Packet waitPacket;
boolean base;
/**
 * Creates a new instance of CollisionDetectionThread
 */
public CollisionDetectionThread(boolean base)
{
  state = IDLE;
  collisionWindow = tempTime = System.currentTimeMillis();
  waitPacket = null;
  this.base = base;
}
/**
 * Thread implementation for run and maintains the states of logic
 * concerning the collision detection algorithm for a full duplex modem.
 */
public void run()
{
   while(true)
   {
      if(System.currentTimeMillis() <= collisionWindow)
      {
         if(base)
         {
            NodeControl.receivingBase = true;
         }
      }
      else
      {
         if(base)
         {
            NodeControl.receivingBase = false;
         }
         if(state == BUSY)
         {
            NodeControl.receiveData(waitPacket);
            state = IDLE;
         }
         else if(state == COLL)
         {
            //drop packet
            System.out.println("Packet Collided(rec) and Dropped from: " +
                        waitPacket.getSrcName());
            state = IDLE;
         }
         else if(state == TRAN)
```

```
            {
                //no received packet had window longer than the transmit window
                state = IDLE;
            }
        }
        yield(); //minimizing busy wait
    }
}
/**
 * Does the overlap checking for all packets and sets the state accordingly.
 * If a packet is involved in a collision, it is dropped and not passed on.
 * @param packet the <code>Packet</code> in question
 */
public long check(Packet packet, long transRate)
{
    //payload size is in bytes(8 bits) and 1000 is to turn into millieseconds
    //duration is in milliseconds
    duration = (long)((float)(packet.getPayloadSize()*8)/transRate*1000);
    System.out.println("Transmission Delay is " + duration);
    if(state == IDLE)
    {
        waitPacket = packet;
        collisionWindow = System.currentTimeMillis() + duration;
        state = BUSY;

    }
    else if((state == BUSY) || (state == COLL))
    {
        tempTime = System.currentTimeMillis() + duration;
        if(tempTime > collisionWindow)
        {
            collisionWindow = tempTime;
            System.out.println("Packet Collided(rec) and Dropped from: " +
                               waitPacket.getSrcName());
            waitPacket = packet;
            //drop original packet that collided
        }
        else //window not longer but still collided
        {
            //drop this packet that collided
            System.out.println("Packet Collided(rec) and Dropped from: " +
                               packet.getSrcName());
        }
        state = COLL;
    }
    else if (state == TRAN)
    {
        tempTime = System.currentTimeMillis() + duration;
        if(tempTime > collisionWindow)
        {
            collisionWindow = tempTime;
            System.out.println("Packet Collided(trans) and Dropped from: " +
                               packet.getSrcName());
            waitPacket = packet; //drop original packet that collided
            state = COLL;
        }
```

```
            else
            {
               System.out.println("Packet Collided(trans) and Dropped from: " +
                                  packet.getSrcName());
            }
         }
         return duration;
      }//end check
      /**
       * need comments
       *
       * not yet called from anywhere
       */
      void setTransmitWindow(long window)
      {
         collisionWindow = window;
         state = TRAN;
      }
}
```

3. Layer2ThreadReceive.java

```
package UAN.Node;

import UAN.Helpers.Payload;
import java.io.IOException;
import java.io.ObjectOutputStream;


/**
 * Handles the communication from <code>NodeControl</code> to the layer 2
 * protocol application over a TCP socket.
 */
public class Layer2ThreadReceive extends Thread
{
   /**
    * OOS used
    */
   private ObjectOutputStream oos;
   /**
    * Constructor
    * @param oos the <code>ObjectOutputStream</code>
    */
   public Layer2ThreadReceive(ObjectOutputStream oos)
   {
      this.oos = oos;
   }//end constructor
   /**
    * Thread implementation for run
    */
   public void run()
   {
      while(true)
      {
         //keep alive
```

```
            yield();//minimizing busy wait
        }
    }//end run
    /**
     * Send the packet to the output stream which is contected to a layer 2
     * protocol.
     * @param payload the <code>Payload</code> to send.
     */
    public void send(Payload payload)
    {
        try
        {
            oos.writeObject(payload);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }//end send
}//end Layer2ThreadReceive
```

4. Layer2ThreadTransmit.java

```
package UAN.Node;

import UAN.Helpers.Packet;
import UAN.Helpers.Payload;
import java.io.IOException;
import java.io.ObjectInputStream;

/**
 * Thread to handle communication between <code>NodeControl</code> and a layer 2
 * protocol application over a TCP socket connection.
 */
public class Layer2ThreadTransmit extends Thread
{
    /**
     * This input stream from the layer 2 protocol.
     */
    private ObjectInputStream ois;
    /**
     * The payload to handle from the layer 2 protocol.
     */
    private Payload payload;
    /**
     * The packet to handle.
     */
    private Packet packet;
    /**
     * used for receivingBase and busy state setting
     */
    private long transmitWindow = System.currentTimeMillis();
    /**
     * The transmission rate of this channel.
     */
    private long transRate;
    private byte[] byteArray;
```

```
    /**
     * Constructor
     * @param ois the <code>ObjectInputStream</code>
     */
    public Layer2ThreadTransmit(ObjectInputStream ois, long transRate)
    {
        this.ois = ois;
        this.transRate = transRate;
    }
    /**
     * Thread implementation for run
     */
    public void run()
    {
        while(true)
        {
            while(NodeControl.receivingBase)
            {
                //can not transmit while receivingBase
                yield();//minimizing busy wait
            }
            try
            {
                NodeControl.transmitingBase = true;
                payload = (Payload) ois.readObject();
                byteArray = (byte[]) payload.getData();
                packet = new Packet(Packet.DATA, payload, NodeControl.baseChannelID);
                packet.setPayloadSize(byteArray.length);
                packet.setSendTime(System.currentTimeMillis());
                NodeControl.transmitData(packet);
                //payload size is in bytes(8 bits) and 1000 is to turn into millieseconds
                //transmitWindow is in milliseconds
                transmitWindow = (long)((float)(packet.getPayloadSize()*8)/transRate*1000);
                try {
                    Thread.sleep(transmitWindow);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                    System.out.println("Did not sleep for duration of transmitWindow");
                }
                NodeControl.transmitingBase = false;
            } catch (IOException ex) {
                ex.printStackTrace();
            } catch (ClassNotFoundException ex) {
                ex.printStackTrace();
            }
            yield();//minimizing busy wait
        }
    }
}
```

C. UAN.HELPERS PACKAGE

1. Address.java

```
package UAN.Helpers;
```

```
import java.io.Serializable;
import java.net.InetAddress;

/**
 * Wraps an IP and port number so that it can be passed as on object in an
 * an input or output stream within the simulation.
 */
public class Address implements Serializable
{
    /**
     * IP address
     */
    private InetAddress IP;
    /**
     * Socket Port number
     */
    private int Port;
    /**
     * Constructor
     * @param ip the <code>InetAddress</code>
     * @param port the <code>Integer</code> representing the port number.
     */
    public Address(InetAddress ip, int port) {

        IP = ip;
        Port = port;
    }
    /**
     * Get the IP address
     * @return the <code>InetAddress</code>
     */
    public InetAddress getIP()
    {
        return IP;
    }
    /**
     * Get the port number.
     * @return the <code>Integer</code> port number.
     */
    public int getPort()
    {
        return Port;
    }

}
```

2 Layer2Connect.java

```
package UAN.Helpers;

import java.io.*;
import java.net.*;

/**
```

```
 * Layer2Connect is an importable helper class for applications that want to
 * connect to the UAN simulation.  These applications are assumed to handle
 * OSI stack functionality of at lease layer 2.  This class will make and
 * maintain the connection to the simulation.  It presents a send() and
 * receive() method calls for layer to layer interface.  This class also
 * performs the data format translation needed from layer 2 to simulation and
 * vice versa.  This class receives and returns a byte array.
 */
public class Layer2Connect implements Serializable
{
   // variables used to establish interface to the Physical Layer simulation
   private static DatagramSocket dgSocket;     // used to request host address
   private static String IP = "192.168.1.1";   // address of Central Host (CH)
   private static int port = 3282;             // datagram port to which CH is listenting
   private static DatagramPacket dgPacket1;    // datagram to PHY requesting host
   private static DatagramPacket dgPacket2;    // datagram from PHY w/address
   private static Socket tcpSocket;            // Interface w/host
   private static Payload payload;
   private static Packet packet;
   private static byte[] byteArray;

   // Object streams through which to exchange wrapped packet
   //with the physical layer simulation
   private static ObjectInputStream ois;
   private static ObjectOutputStream oos;
   // input stream with which to interface with the application user
   static BufferedReader br;
   /**
    * This class is a helping class for applications to request a connection to
    * the UAN Simulation.  This application is expected to act as a layer 2
    * process. This class is offered as an importable class that can be
    * instantiated in order to establish communications with the simulation.
    * It is intended to completely abstract the connection implementation but
    * uses TCP sockets.
    */
   public Layer2Connect () throws UnknownHostException,
                                                    IOException
   {
      br = new BufferedReader(new InputStreamReader(System.in));
      // Establish interface to Servicing "modem" Host
      // 1. Get Central Host (CH) IPAddress and port number from user
      System.out.println("Enter CentralHost IP: <enter for default> ");
      IP = br.readLine().trim();
      if(IP.length()<7)
      {
            IP = "192.168.1.1"; //default CH
      }
      System.out.println("Enter CentralHost port: <enter for default> ");
      try{
            port = Integer.parseInt(br.readLine().trim());
      }catch (Exception e)
      {
            port = 3282; //default port
      }
      dgSocket = new DatagramSocket();
      System.out.println("dgSkt port "+
```

```
            dgSocket.getLocalPort()+" "+ dgSocket.getPort());

        // - empty UDP packet serves as query to CH
        dgPacket1 = new DatagramPacket(
              new byte[0], 0, InetAddress.getByName(IP), port);
        dgSocket.send(dgPacket1);

        // - receive response from CH and extract host information
        dgPacket2 = new DatagramPacket(new byte[0], 0);
        dgSocket.receive(dgPacket2);
        System.out.println("Servicing \"modem\" host: "+
              dgPacket2.getAddress() +
              " Port: " + dgPacket2.getPort());

        // instantiate a TCP connection using the query information
        tcpSocket = new Socket(dgPacket2.getAddress(), dgPacket2.getPort());

        // instantiate i/o stream to sent/receive information
        //through the "modem" host
        oos = new ObjectOutputStream(tcpSocket.getOutputStream());
        ois = new ObjectInputStream(tcpSocket.getInputStream());
    }
    /**
     * This method is the interface for layer 2 to pass data to the
     * simulation.
     * @param in is a byte array from the upper layer (assumed layer 2)
     */
    public void send(byte[] in)
    {
        payload = new Payload(in);
        try {
            oos.writeObject(payload);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    /**
     * This method is the interface for layer 2 to get data from the simulation.
     * @return byteArray a byte array is returned
     */
    public byte[] receive()
    {
        try {
            payload = (Payload) ois.readObject();
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        byteArray = (byte[]) payload.getData();
        return byteArray;
    }
}
```

3. Packet.java

```
package UAN.Helpers;

import java.io.Serializable;
/**
 * This <code>Pakcet</code> is the data the packet that is passed within the UAN
 * simulation.  Its data types have bearing on the different functions with the
 * logic of <code>NodeControl</code> and how each packet is handled.
 */
public class Packet implements Serializable
{
    /**
     * Packet type if payload received from layer 2 protocol.
     */
    public final static int DATA       = 11;
    /**
     * Packet type only sent from <code>SimControl</code> to define nodes.
     */
    public final static int NODEINFO   = 22;
    /**
     * Packet type only sent from <code>SimControl</code> to initiate
     * a connection to the defined layer 2 protocol.
     */
    public final static int LAYER2     = 33;
    /**
     * Packet type only sent from the <code>TestSend</code> class mainly for
     * testing.
     */
    public final static int SIM        = 44;
    /**
     * The packet type of this packet.
     */
    private int packetType;
    /**
     * The payload size within this packet.
     */
    private int payloadSize;
    /**
     * The time this packet was sent.  Used to calculated simulated propogation
     * delay.
     */
    private long sendTime;
    /**
     * The payload contained withing this packet.
     */
    private Payload payload;
    /**
     * String names for destination and source node name
     */
    private String destName, srcName;
    /**
     * Channel that this packet transmitted in
     */
    private int channelID;
    /**
```

```
     * THIS CONSTRUCTOR USED DURING DEVELOMENT, TESTING AND SETUP.
During
     * simulation runs, the destintation will not be a function of this layer.
     * Only the source will matter in this simulation so that broadcast
     * neighborhoods can be determined.
     * @param type the <code>Integer</code> representation of packet type
     * @param payload the <code>Payload</code> in this packet
     * @param destination the destination name as a String
     * @param channel the channel ID
     */
    public Packet(int type, Payload payload, String destination, int channel)
    {
       packetType      = type;
       this.payload    = payload;
       destName        = destination;
       channelID       = channel;
       payloadSize     = 10; //bytes :: default
    }
    /**
     * This constructor used for actual environment simulation runs.
     * @param type the <code>Integer</code> representation of packet type.
     * @param payload the <code>Payload</code> in this packet.
     * @param channel the <code>Channel</code> this packet was transmitted on.
     */
    public Packet(int type, Payload payload, int channel)
    {
       this(type, payload, null, channel);
    }
    /**
     * Gets the channel ID
     * @return the channel ID as an integer
     */
    public int getChannelID()
    {
       return channelID;
    }
    /**
     * Sets the source node name of this packet
     * @param s the source name
     */
    public void setSrcName(String s)
    {
       srcName = s;
    }
    /**
     * Gets the source name.
     * @return the name of the node that sent this packet.
     */
    public String getSrcName()
    {
       return srcName;
    }
    /**
     * Gets the payload from this packet.
     * @return the payload carried by this packet as a UAN <code>Payload</code> object.
     */
```

```
    public Object getPayload()
    {
       return payload;
    }
    /**
     * Gets the payload size carried by this packet.
     * @return the payload size in this packet.
     */
    public int getPayloadSize()
    {
       return payloadSize;
    }
    /**
     * Sets the payload size carried by this packet.
     * @param numBytes is the number of bytes that this payload carries.
     */
    public void setPayloadSize(int numBytes)
    {
       payloadSize = numBytes;
    }
    /**
     * Get this packet's type.
     * @return this packet's type
     */
    public int getPacketType()
    {
       return packetType;
    }
    /**
     * Sets the send time for this packet.
     * @param time the send time milliseconds
     */
    public void setSendTime(long time)
    {
       sendTime = time;
    }
    /**
     * Gets the sending time of this packet.
     * @return the send time of this packet in milliseconds.
     */
    public long getSendTime()
    {
       return sendTime;
    }
    /******** Testing Methods *****************************************/

    /**
     * Gets the destination node name for this packet.  Only set and used in
     * testing and demonstrations of logic.
     * @return the destination node name.
     */
    public String getDestName()
    {
       return destName;
    }
}//end packet
```

4. Payload.java

```
package UAN.Helpers;

import UAN.Sim.NodeInfo;
import java.io.Serializable;

/**
 * Used as a wrapper for any payload defined.  Abstracts for the rest of the
 * simulation.  Payload type only need to be defined here.
 */
public class Payload implements Serializable
{
    Object o;
    /**
     * String Constructor
     * @param s the <code>String</code> to be the payload.
     */
    public Payload(String s) {
        o = s;
    }
    /**
     * NodeInfo Constructor
     * @param node the <code>NodeInfo</code> to be the payload.
     */
    public Payload(NodeInfo node) {
        o = node;
    }
    /**
     * Integer Constructor.
     * Converted to a String.
     * @param num the <code>Integer</code> to be the payload.
     */
    public Payload(int num)
    {
        String snum = Integer.toString(num);
        o = snum;
    }
    /**
     * Address Constructor.
     * @param address the <code>Address</code> to be the payload.
     */
    public Payload(Address address)
    {
        o = address;
    }
    /**
     * Byte Array constructor used by the simulated layer 2 application.
     * @param myByte the <code>byte[]</code> to be the payload.
     */
    public Payload(byte[] in)
    {
        o = in;
    }
```

```
    /**
     * Gets the object that is payload.
     * @return the object in the payload.
     */
    public Object getData()
    {
        return o;
    }
}
```

5. UAN_Net.dtd

```
<?xml version="1.0" encoding="UTF-8"?>

<!--

-->
<!-- PCDATA stands for Parsed Character Data and it means that the element contains text data " -
->
<!ELEMENT NodeName (#PCDATA)>
<!ELEMENT NodeType (#PCDATA)>
<!ELEMENT NodeLocation (#PCDATA)>
<!ELEMENT NodeDepth (#PCDATA)>
<!ELEMENT ChannelID (#PCDATA)>
<!ELEMENT ChannelName (#PCDATA)>
<!ELEMENT Channel (ChannelName, ChannelID)>
<!ELEMENT Node (NodeName, NodeType, NodeLocation, NodeDepth, Channel+)>
<!ELEMENT UAN_Net (Node+)>
```

6. UAN4node.xml example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE UAN_Net SYSTEM "UAN_net.dtd">
<UAN_Net>
    <Node>
        <NodeName>NodeA</NodeName>
        <NodeType>Static</NodeType>
        <NodeLocation>32 -96</NodeLocation>
        <NodeDepth>10</NodeDepth>
        <Channel>
            <ChannelName>NodeAbase</ChannelName>
            <ChannelID>0</ChannelID>
        </Channel>
        <Channel>
            <ChannelName>NodeAsec1</ChannelName>
            <ChannelID>1</ChannelID>
        </Channel>
    </Node>
    <Node>
        <NodeName>NodeB</NodeName>
        <NodeType>Static</NodeType>
        <NodeLocation>32 -96.01</NodeLocation>
        <NodeDepth>10</NodeDepth>
        <Channel>
```

```
            <ChannelName>NodeBbase</ChannelName>
            <ChannelID>1</ChannelID>
        </Channel>
        <Channel>
            <ChannelName>NodeBsec1</ChannelName>
            <ChannelID>0</ChannelID>
        </Channel>
        <Channel>
            <ChannelName>NodeBsec2</ChannelName>
            <ChannelID>2</ChannelID>
        </Channel>
    </Node>
    <Node>
        <NodeName>NodeC</NodeName>
        <NodeType>Static</NodeType>
        <NodeLocation>32 -96.015</NodeLocation>
        <NodeDepth>10</NodeDepth>
        <Channel>
            <ChannelName>NodeCbase</ChannelName>
            <ChannelID>0</ChannelID>
        </Channel>
        <Channel>
            <ChannelName>NodeCsec1</ChannelName>
            <ChannelID>1</ChannelID>
        </Channel>
        <Channel>
            <ChannelName>NodeCsec2</ChannelName>
            <ChannelID>2</ChannelID>
        </Channel>
    </Node>
    <Node>
        <NodeName>NodeD</NodeName>
        <NodeType>Static</NodeType>
        <NodeLocation>32.008 -96.01</NodeLocation>
        <NodeDepth>10</NodeDepth>
        <Channel>
            <ChannelName>NodeDbase</ChannelName>
            <ChannelID>2</ChannelID>
        </Channel>
        <Channel>
            <ChannelName>NodeDsec1</ChannelName>
            <ChannelID>0</ChannelID>
        </Channel>
        <Channel>
            <ChannelName>NodeDsec2</ChannelName>
            <ChannelID>1</ChannelID>
        </Channel>
    </Node>
</UAN_Net>
```

D. UAN.CHANNELS PACKAGE

1. Channel.java

```
package UAN.Channels;
```

```
import java.io.Serializable;

/**
  * Superclass for all channels with each instantiation representing a set of
 * atrributes that defines a transmit and receive capability and limitation.
 */
public class Channel implements Serializable
{
    /**
     * Constant for kilobits per second
     */
    static final int Kbps = 1000;
    /**
     * Transmision rate for this channel
     */
    private int transmitRate;
    /**
     * Frequency for this channel
     */
    private int frequency;
    /**
     * User designated name for this channel
     */
    private String name = null;
    /**
     * Unique ID for this channel
     */
    private int ID;
    /**
     * Constructor
     * @param name is the user asigned name of this channel.
     * @param transmitRate is the tranmission rate of this channel.
     * @param frequency is the frequency of this channel.
     * @param ID is the unique channel ID of all channels defined.
     */
    public Channel(String name, int transmitRate, int frequency, int ID)
    {
        this.name          = name;
        this.transmitRate  = transmitRate;
        this.frequency     = frequency;
        this.ID            = ID;
    }
    /**
     * gets the transmission rate of this channel.
     * @return the transmission rate.
     */
    public int getTransmitRate() {
        return transmitRate;
    }
    /**
     * gets the frequency of this channel.
     * @return the frequency.
     */
    public int getFrequency() {
        return frequency;
    }
```

```
    /**
     * gets the user asigned name of this channel.
     * @return the discriptive name of this channel set by the user.
     */
    public String getName() {
        return name;
    }
    /**
     * get the unique ID of this channel compared to all defined channels.
     * @return the unique ID of this channel.
     */
    public int getID()
    {
        return ID;
    }
}
```

2. ChannelZero.java

```
package UAN.Channels;

/**
 * Instantiation of ChannelZero
 */
final public class ChannelZero extends Channel
{
    /**
     * Constructor
     */
    public ChannelZero(String name)
    {
        super(name, 1*Kbps, 4, 0);
    }
}
```

3. ChannelOne.java

```
package UAN.Channels;

/**
 *
 * Instantiation of ChannelOne
 */
final public class ChannelOne extends Channel
{
    /**
     * Constructor
     */
    public ChannelOne(String name)
    {
        super(name, 1*Kbps, 1, 1);
    }
}
```

4. ChannelTwo.java

```
package UAN.Channels;

/**
 *Instatiation of ChannelTwo
 */
final public class ChannelTwo extends Channel
{
    /**
     * Constructor
     */
    public ChannelTwo(String name)
    {
        super(name, 1*Kbps, 6, 2);
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Kilfoyle, D. B. and Baggeroer, A. B.. *The State of the Art in Underwater Acoustic Telemetry*. IEEE Journal of Oceanic Engineering, OE-25(5):4--27, January 2000.

[2] Mclaughlin, Brett. (n.d.)
"Cornelius Drebbel: inventer of the submarine." Retrieved August 10, 2006, from http://www.dutchsubmarines.com/specials/special_drebbel.htm

[3] Pike, John E. (April 26, 2005)
Sound Surveillance System. Retrieved August 19, 2006, from http://www.globalsecurity.org/intell/systems/sosus.htm

[4] Akyildiz, I. F., Pompili, D. and Melodia, T. *Underwater Acoustic Sensor Networks: Research Challenges*. Ad Hoc Networks Jounal (Elseviier), March 2005, pp. 257-279.

[5] Proakis, J. G. , Sozer, E. M. , Rice, J. A. and Stojanovic, M. Shallow Water Acoustic Networks, IEEE Communication Magazine, Vol. 39, No. 11, pp. 114-119, November 2001

[6] Rice, J. A., Creber, R. K., Fletcher, C. L., Baxley, P. A., Davison, D. C. and Rogers, K. E., Seaweb Undersea Acoustic Nets, Biennial Review 2001, SSC San Diego Technical Document TD 3117, pp. 234-250, August 2001

[7] Benton, C., Kenny, J., Nitzel, R., Blidberg, D. R., Chappell, S. G., Mupparapu, S. S., In *IEEE/OES AUV2004: A Workshop on Multiple Autonomous Underwater Vehicle Operations*, Sebasco Estates, Maine, June, 2004.

[8] Gibson, John, Xie, Geoffrey, Yang Wen-Bin, "Design and Implementation of a Distributed Time-Driven UAN Modeling System: for Jointly Evaluating Signal Processing Techniques and Acoustic Networking Protocols," Unpublished Manuscript March 2006.

[9] Zingarelli, R.A. and King, D.B., "RAM to Navy Standard Parabolic Equation: Transition from Research to Fleet Acoustic Model," 2003 NRL Review , NRL website:

[10] Smith, Kevin B., "Convergence, Stability, and Variability of Shallow Water Acoustic Predictions Using a Split-Step Fourier Parabolic Equation Model," J. Comp. Acoust., Vol. 9, No. 1, pp. 243-285, 2001 - Special Issue of Proceedings of the Shallow Water Acoustic Modeling (SWAM'99) Workshop, 8 - 10 September 1999 (eds. Alex Tolstoy and Kevin B. Smith)

[11] Smith, Kevin B. and Tappert, Frederick, "Monterrey-Miami Parabolic Equation," Release Documentation.

[12] Sozer, Ethem M., Stojanovic, Milica and Proakis, John G., "Design and Simulation of an Underwater Acoustic Local Area Network," Northeastern University, Communications and Digital Signal Processing Center, 409 Dana Research Building, Boston, Massachusetts, 1999.

[13] Coelho, Jose, "Underwater Acoustic Networks: Evaluation of the Impact of Media Access Control on Latency, in a Delay Constrained Network," Master's Thesis (MS-CS), Naval Postgraduate School, Monterey, California, March 2005.

[14] Gibson, John, Xie, Geoffrey, Coelho, José and Diaz-Gonzalez, "The Impact of Contention Resolution verses a priori Channel Allocation Delay Constrained Network," Undersea Networking (Unet) Workshop, in support of The Technical Cooperative Program (TTCP), Monterey, CA, January 2005.

[15] L Diaz-Gonzalez, "Underwater Acoustic Networks: An Acoustic Propagation Model for Simulation of Underwater Acoustic Networks," Master's Thesis (MS-CS), Naval Postgraduate School, CA, December 2005

[16] Horner, Doug (2006) Personal conversation at the Naval Postgraduate School regarding the actual use and limitations of acoustic modems being tested, June.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California